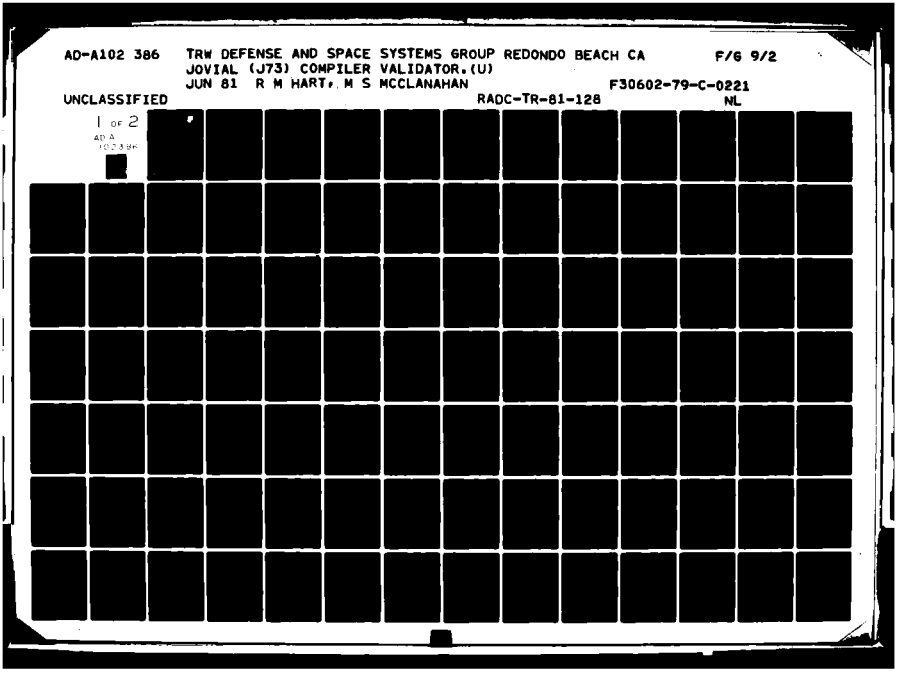
AD-A102 386 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CA F/G 9/2
JOVIAL (J73) COMPILER VALIDATOR.(U)
JUN 81 R M HART, M S MCCLANAHAN F30602-79-C-0221
UNCLASSIFIED RADC-TR-81-128 NL

1 OF 2
AD A
102386



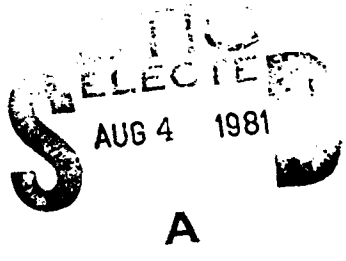
LEVEL II (12)

RADC-TR-81-128
Final Technical Report
June 1981

JOVIAL (J73) COMPILER VALIDATOR

TRW Defense & Space Systems Group

Ruth M. Hart
Marilyn S. McClanahan



ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-128 has been reviewed and is approved for publication.

APPROVED: Clement D. Falzarano

CLEMENT D. FALZARANO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-128	2. GOVT ACCESSION NO.: AD-A102386
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): JOVIAL (J73) COMPILER VALIDATOR	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, Aug 1979 — Mar 1981
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): Ruth M. Hart, Marilyn S. McClanahan	8. CONTRACT OR GRANT NUMBER(s): F30602-79-C-0221
9. PERFORMING ORGANIZATION NAME AND ADDRESS: TRW Defense and Space Systems Group, One Space Park, Redondo Beach CA 90278	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 25320201
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: June 1981
	13. NUMBER OF PAGES: 102
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Clement Falzarano (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL, JOVIAL (J73), MIL-STD-1589A, Compiler Testing, Compiler Validation, Validator, SEMANOL, SEMANOL (76), interpreter, Test Effectiveness Measurement Facility, software measurement, software quality, test effectiveness, computer programs, software.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the results of a project whose goal was to develop a methodology for the systematic testing of compilers. Three major products were developed: a compiler validator for the JOVIAL (J73) programming language, a SEMANOL (76) specification of JOVIAL (J73), and a Test Effectiveness Measurement Facility. In addition, the SEMANOL (76) Interpreter was modified to support this effort. The programming language JOVIAL (J73) was extensively analyzed. Appendix A contains documentation

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED (Cont'd)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4-107

Item 20 (Cont'd)

of the SEMANOL specification, Appendix B contains a listing of the SEMANOL specification, and Appendix C contains revised documentation for the SEMANOL Interpreter. Additional details on the Validator and the Test Effectiveness Measurement Facility can be found in their respective Users Manuals.

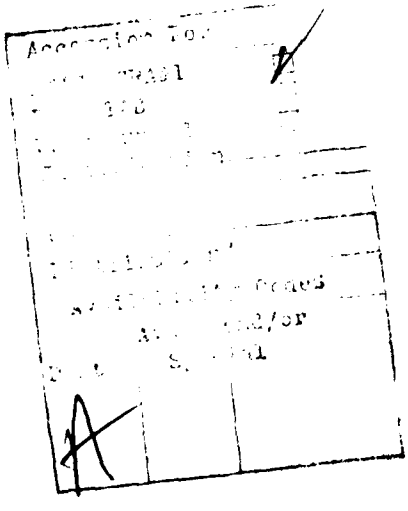


TABLE OF CONTENTS

INTRODUCTION

In 1976-1977, TRW performed a research study (Contract F30602-76-C-0255) for Rome Air Development Center (RADC) on methods of Formal Compiler Testing. This study investigated the nature of compiler testing, the potential advantages to be realized through automation of the process, and the use of SEMANOL, TRW's metalanguage for describing the syntax and semantics of programming languages, as the basis for such automation. A design was produced for a semi-automatic test generation system that could be implemented with some further research.

When the current project was first proposed in January, 1978, with the research goal of developing a methodology for systematic testing of compilers, the intent was to design a SEMANOL-based methodology for measuring the quality of a validator and a method of using these measurements to guide validator improvement; that is, it was envisioned as a logical counterpart to the prior project. Test cases were to be generated during the course of the project, but they were not expected to be important in and of themselves; rather, their importance was as input to the proposed "Test Effectiveness Measurement Facility". In fact, the possibility of using the automated test generation tools designed in the previous contract to create the test cases was considered. JOVIAL(J73) (actually J73/I) was proposed as the prototype language. In 1975, TRW had produced a SEMANOL specification of this language for RADC (Contract F30602-76-C-0238). This SEMANOL specification would have to be updated, as both the SEMANOL and JOVIAL(J73/I) languages had since been modified (to SEMANOL(76) and MIL-STD-1589, respectively), but these revisions were expected to be minor.

By the time the contract for this project was actually awarded in August, 1979, the circumstances had changed substantially. JOVIAL(J73/I) had been merged with another dialect of JOVIAL, JOVIAL(J3B), to produce a new JOVIAL(J73), defined in MIL-STD-1589A; although this new language retained its old name, it was actually a very different language. This had two major implications for the project. First, it meant that the original SEMANOL specification of JOVIAL(J73) could no longer be used; instead, an entirely new specification would have to be written. Secondly, the test cases themselves assumed a new

importance; they could be used to test conformance of compilers for the new language to MIL-STD-1589A. As a result, the development and use of the Test Effectiveness Measurement Facility evolved naturally into a minor part of the project, rather than its major focus.

A second change occurred shortly before the start of the project; namely, RADC began charging for use of their H-6180 Multics computer. Up until this time, unlimited Multics resources had always been available for TRW's SEMANOL projects; for this project, on the other hand, only $50,000, with a maximum of 2900 blocks of file storage, could be allocated. These resources would not be nearly sufficient to run the set of JOVIAL(J73) test cases through the SEMANOL Interpreter, as had been planned; the SEMANOL Interpreter system is extremely slow, and the output from the Interpreter is quite bulky. This presented several problems. If the SEMANOL specification could not be used for debugging the test set, another method had to be found. Since the JOVIAL(J73) JOCIT compiler was scheduled for delivery about the time this project was to begin, it was decided in contract negotiations that it, rather than the SEMANOL specification, should be used to debug the Validator. The JOCIT compiler is hosted on an IBM 360/370, and it would be run on a TRW computer; thus it made little sense to develop the test cases at RADC, and then transport them to TRW. In addition, if the entire test set could not be processed through the SEMANOL specification, it would not be possible to determine its quality using the Test Effectiveness Measurement Facility. Thus, the Measurement Facility took on an even smaller role.

Thus, when the project began, its aims had changed significantly. Although the overall goal remained the same, that is, to develop a methodology for the systematic testing of compilers, there was now considerably more interest in the products (especially the Validator itself) and less interest in the research. In approximate order of importance, the goals were now:

- To develop a well-designed set of JOVIAL(J73) programs which would comprehensively test conformance of JOVIAL(J73) compilers to MIL-STD-1589A;
- To write a SEMANOL specification of JOVIAL(J73);

- To develop a facility for measuring the quality of test sets for any language, given the existence of a SEMANOL specification for that language.

It should be noted that the connection between these three tasks had become rather tenuous. Although the process of writing the SEMANOL specification would suggest difficult areas of the language and thus possible tests, and though the design of the Test Effectiveness Measurement Facility would place constraints on the format of the SEMANOL specification, the three tasks had become essentially independent.

The next section of this report describes the results of the comprehensive analysis of JOVIAL(J73) that was performed as a prerequisite for writing both the test cases and the SEMANOL specification. Subsequent sections describe the Validator, the SEMANOL specification, and the Test Effectiveness Measurement Facility. The final section summarizes the achievements of this project and the problems that were encountered, and suggests directions for the future.

ANALYSIS OF JOVIAL(J73)

A detailed analysis of the JOVIAL(J73) language was undertaken prior to construction of both the Validator test cases and the SEMANOL specification of JOVIAL(J73). The major goal of this analysis was identification of potential problem areas of the language - difficult-to-implement features as well as ambiguities - which should be especially well-tested in a Validator. In addition, in order to describe the semantics of a programming language in SEMANOL, it is necessary to fully understand the semantics of that language. Thus, a secondary goal of the analysis was to become JOVIAL(J73) "experts".

Although TRW was very familiar with JOVIAL(J73/I), having performed two prior analyses of that language, there were many changes and several entirely new features in the revised JOVIAL(J73) language. Furthermore, JOVIAL(J73) did not go through a rigorous design phase, as did Ada. Rather, it was created in a very short time frame, essentially as a merger of two existing dialects of JOVIAL, J73/I and J3B. There was little time to determine how the two languages interacted, and, in particular, whether there were areas of conflict. Therefore, it was especially important to examine the new features of the language, such as fixed point arithmetic, strong typing, and pointers.

During the course of this project, several project members have become very active on the Language Issues Committee of the JOVIAL Users Group, and our analysis has largely been performed in the context of that committee. TRW is one of only three companies other than the major compiler developers to be regularly represented on the Committee; our perspective, as language experts who are not also compiler writers, is unique. Since August, 1979, the Language Issues Committee has considered almost 200 Language Change Requests, and has accepted nearly 100 LCRs. The relationship of this project to the Committee's work has been mutually beneficial:

- We were able to determine, by examining the early Language Change Requests, which areas of JOVIAL(J73) were likely to contain ambiguities and require changes; this information guided our efforts both in writing test cases and developing a SEMANOL specification.

- We were able to receive first-hand information from the language designers on how certain ambiguities should be resolved.

- We have carefully scrutinized all Language Change Requests submitted to the Committee and have helped to decide whether they should be approved. For example, many of the more recent Language Change Requests have proposed major rewordings to MIL-STD-1589A/B, which sometimes inadvertently change the meaning of the standard; we have often been able to detect such subtle shifts in meaning, and thus we have played an important role in stabilizing the language.

- Members of the project have themselves proposed 38 Language Change Requests, of which some have been accepted, some have been rejected for a variety of reasons, and many have not yet been considered. Those which have been approved or which are still pending are summarized later in this section.

TRW's Language Change Requests

The Language Change Requests (LCRs) that have been presented to the Language Issues Committee of the JOVIAL Users Group by the members of this project are summarized below. Some pointed out ambiguities in JOVIAL(J73) that needed to be resolved before either a formal semantic specification could be written or test cases generated for the pertinent sections of the language. Other LCRs identified deficiencies in the language and proposed solutions; some of these were rejected because they were not upward compatible with MIL-STD-1589A, rather than because they lacked merit.

All of the Language Change Requests which have been adopted or are still pending are described here, with the exception of four LCRs (TRW #1, TRW #10, TRW #24, and TRW #36) which point out typos in MIL-STD-1589A or MIL-STD-1589B. In addition, those which were superseded by other LCRs or which were dropped for reasons of non-upward-compatibility are also included. For each LCR, the following information is provided:

- Title and Number (TRW #, LCR # if assigned)
- Statement of Issue
- Proposed Resolution (summary)
- Status as of January, 1981

1. Semantics of Parallel Tables (TRW #2, LCR L140.1)

   Issue: A prior change to MIL-STD-1589A (LCR L65.1) caused one constraint to be redundant and created a new constraint in a Semantics section of the standard.

   Resolution: Delete the redundant constraint; move the new constraint from Semantics to Constraints.

   Status: The redundant constraint was retained; the other constraint was moved; incorporated into MIL-STD-1589B.

2. Storage of Non-Tight Tables (TRW #3, LCR L141.1)

   Issue: The meaning of the absence of a <structure-specifier> T for a table is not sufficiently defined.

   Resolution: Add a new semantics paragraph to Section 2.1.2.2 of MIL-STD-1589A.

   Status: Approved, with slightly modified wording; incorporated into MIL-STD-1589B.

3. Order of Expansion of Define Calls (TRW #4, LCR L142)

   Issue: The order of expansion of define-calls is insufficiently defined.

   Resolution: Add text to the Semantics section of Section 2.4.1 of MIL-STD-1589A.

   Status: Dropped; the issue raised is resolved by the approval of LCR L18.3.

4. Filler Bits (TRW #5, LCR L143.1)

   Issue: The definition of "filler bits" for bit and rep-conversions is not complete or self-consistent.

   Resolution: Add wording to Section 7.0 to clarify the meaning of "filler bits".

   Status: Deferred; however, the Language Control Facility agrees that the proposed changes are an accurate re-phrasing of the intent of MIL-STD-1589A/B.

5. Declared vs. Implemented Sizes of Items (TRW #6, LCR L144.1)

   Issue: The freedoms and constraints for implemented sizes of some data items are not consistently delineated.

   Resolution: Add notes to Sections 2.1.1, 2.1.2.3, and 2.1.2.4 to improve the clarity of MIL-STD-1589A/B.

   Status: Deferred; with minor exceptions, the Language Control Facility agrees that the proposed changes are accurate clarifications of MIL-STD-1589A/B.

6. Compiler Obligations for Reordering and Sizing (TRW #7, LCR L145.1)

   Issue: Compiler freedoms and constraints are inconsistently described.

   Resolution: Add text to Section 2.1, and modify Sections 2.1.2.3 and 9.11 to improve the clarity of MIL-STD-1589A/B.

   Status: Deferred; however, the Language Control Facility agrees that the proposed change is an improvement over the existing text.

7. Storage of Character Items (TRW #9, LCR L147.1)

   Issue: The storage of character items in tables is not consistently constrained between ordinary dense tables and specified tables.

   Resolution: Change the semantics of ordinary dense tables to agree with the semantics of specified tables in that respect.

   Status: Approved; incorporated into MIL-STD-1589B.

8. Item Positioning in Dense Tables (TRW #12, LCR L170.1)

   Issue: Allocation of items in a dense table should be more specifically defined, to prohibit certain inefficient implementations and to provide meaning to otherwise vacuous cases.

   Resolution: Insert text in Section 2.1.2.3 of MIL-STD 1589 A/B to insure left-justification of dense-packed items.

   Status: Deferred.

9. Entry Positioning in Tight Tables (TRW #13, LCR 171.1)

Issue: The definition of tight structure does not adequately prohibit certain inefficient implementations.

Resolution: Insert text in Section 2.1.2.2 of MIL-STD 1589A/B to prohibit such inefficiencies.

Status: Deferred.

10. DEFINE List-Options (TRW #15)

Issue: The placement of the list options within DEFINE-declarations is not sufficient to assure their intended interpretation.

Resolution: Precede each option by an exclamation mark.

Status: Withdrawn before submission to the Language Issues Committee; the issue raised is resolved by LCR L73.3.

11. DEFINEs, SKIPs, COPYs, and Equivalence of Upper/Lowercase Letters (TRW #16, LCR L173.1)

Issue: Expansion of define-calls and text directives interacts with equivalence of uppercase and lowercase letters.

Resolution: Replace text in Section 8.1 of MIL-STD 1589A/B.

Status: Concept approved; alternate wording adopted.

12. SKIP Directives (TRW #17, LCR L174.1)

Issue: The meanings of !SKIP, !BEGIN, and !END are not well-defined.

Resolution: Replace text in Section 9.2.2 of MIL-STD 1589A/B.

Status: Concept approved; alternate wording adopted.

13. Items Right-Justified in Entries (TRW #18, LCR L175.1)

Issue: The intended meaning of entries being right-justified needs to be defined.

Resolution: Add text to Section 2.1.2.2 of MIL-STD-1589A/B.

Status: Approved.

14. Sizes of BIT/BYTE Function Values (TRW #20, LCR L177)

Issue: Bit/character sizes of values returned by the intrinsic BIT/BYTE functions in formulas are different from the sizes of those functions in pseudovariables.

Resolution: Change the size returned by the BIT/BYTE functions in formulas to the size returned by those functions in pseudovariables.

Status: Rejected; although desirable, the change is not upward-compatible. LCR L158.3, which has been approved by the JOVIAL Users Group but not by the Language Control Board, introduced the SUBSTR function to provide the desired capability in an upward-compatible manner.

15. Other Ambiguous Status Constants (TRW #21, LCR L178.1)

Issue: Some of the contexts which obligate a compiler to implicitly disambiguate a status-constant are not constrained to ensure that this is possible.

Resolution: Add text to Sections 2.1.2.1 and 7.0 of MIL-STD 1589A/B to resolve the problem.

Status: Approved, with some minor wording changes.

16. No Truncation of Bit Formulas in While Clauses (TRW #22, LCR L179.1)

Issue: The <while-clause> of a WHILE <loop-statement> was inadvertently omitted from LCR L83 (which prohibits implicit conversion of <bit-formulas> to <boolean-formulas> in <while-phrases>, <if-statements>, and <trace-controls>).

Resolution: Add text to Sections 4.2 and 5.2.2 of MIL-STD-1589A/B to correct the situation.

Status: Approved by JOVIAL Users Group and Language Control Facility; deferred by Language Control Board - although an oversight, it is not upward-compatible.

17. Define-calls Produce Complete Symbols (TRW #25)

Issue: Section 2.4.1, constraint 3, of MIL-STD-1589B describes how the compiler must recognize symbols (i.e., the end of a substituted define string terminates any symbol), rather than how program text is constrained.

Resolution: Replace the offending constraint and add new text to the Semantics portion of Section 2.4.1, MIL-STD-1589B.

Status: Not yet considered.

18. Quotes in Actual-Define-Parameters (TRW #26)

Issue: When does an <actual-define-parameter> contain enclosing quotes?

Resolution: Make the text in Section 2.4.1 of MIL-STD-1589B more explicit.

Status: Not yet considered.

19. More on REF <statement-name> (TRW #27)

Issue: Correct an oversight to LCR L118.

Resolution: Add words to the constraint in Section 2.3 of MIL-STD-1589B.

Status: Not yet considered.

20. Scope of Names (TRW #28)

Issue: Section 1.3 of MIL-STD-1589B does not provide a sufficiently clear, complete, and consistent description of the scopes of names in a JOVIAL(J73) program.

Resolution: Rewrite Section 1.3 of MIL-STD-1589B.

Status: Not yet considered.

21. Abort-Phrase Semantics (TRW #29)

Issue: Does an <abort-phrase> take effect at the beginning of the execution of the <procedure-call-statement> which contains it, or only at the beginning of the execution of the body of the procedure being invoked?

Resolution: To be determined.

Status: Not yet considered.

22. Subscript-Index Type Constraints (TRW #30, LCR L194.1)

Issue: The type of a dimension is not well-defined and subscript-index type constraints are too restrictive.

Resolution: Add an exact definition of the type of a dimension, and require indices to have implicitly-convertible types, rather than equivalent types.

Status: Approved by JOVIAL Users Group; not yet considered by Language Control Board.

23. Type-Equivalence of *-Dimension Tables (TRW #31)

Issue: Type-equivalence is not fully defined for *-dimension tables.

Resolution: Add text to Section 7.0 of MIL-STD-1589B.

Status: Not yet considered.

24. Assigning to Substrings of Function Return-Values (TRW #32)

Issue: Assignments to a BIT or BYTE substring of a function return-value (denoted by the <function-name>) are not allowed in MIL-STD-1589B, but were allowed in MIL-STD-1589A.

Resolution: Change the wording of Sections 3.2 and 6.1 of MIL-STD-1589B to permit this situation.

Status: Not yet considered.

25. Identity of Automatic Data Objects (TRW #35)

Issue: The current wording of Section 2.1.5, MIL-STD-1589B, implies a particular implementation strategy.

Resolution: Rewrite Section 2.1.5 of MIL-STD-1589B to be implementation-independent.

Status: Not yet considered.

26. Type of LOC (<block-dereference>) (TRW #34)

Issue: It is unclear whether LOC (<block-dereference>) is a typed or untyped pointer; this was an oversight in LCR L95.

Resolution: The pointer should be typed.

Status: Not yet considered.

27. Status-Type Enumeration (TRW #35, LCR L193)

Issue: Status types cannot conveniently be used for loop iterations in a well-structured programming style.

Resolution: Add an <until-phrase> analogous to a <while-phrase>, but which tests the control-variable before modifying it.

Status: Deferred.

28. Referencing Components of Nested Blocks (TRW #37)

Issue: It is not currently possible to reference components of a nested block when both the outer and inner blocks have named types.

Resolution: Add syntax to Section 6.3.1 of MIL-STD-1589B to permit such references.

Status: Not yet considered.

29. Semantics of Table Assignment (TRW #38)

Issue: According to the semantics of MIL-STD-1589B, table assignment cannot be implemented using reference semantics in all cases.

Resolution: Do the existing compilers conform to these semantics? If not, perhaps the semantics should be changed.

Status: Not yet considered.

Summary

The main conclusion of our analysis is that JOVIAL(J73) is an extremely large and complex language that is difficult to learn. This view is substantiated by the fact that in the two years after the design of JOVIAL(J73) was completed, over 200 Language Change Requests have been submitted to the Language Control Facility, many by the compiler developers themselves. In addition, a large proportion of the problems submitted to the compiler developers as compiler errors turn out to be programmer errors instead.

There are many areas of JOVIAL(J73) in which users are likely to misinterpret the semantics of the language. For example, the introduction of strong typing into an already existing language prohibited many common programming constructs such as those described below:

1. Because of the strong type-checking rules, and the fact that a table-entry was always considered to be of type "table", it was not possible, in MIL-STD-1589A, to use bodiless tables as arrays, as was permitted in JOVIAL(J73/I). JOVIAL(J3B), which had strong typing, also had an ARRAY data structure, but this construct was not included in JOVIAL(J73) either. As a result, the only way to use an array in JOVIAL(J73) was to declare a table with a single-item body, a clumsy and unnatural solution.

   EX : Given the table declaration

   ```
   TABLE TT(10) S ;
   ```

   one could not write

   ```
   TT(3) = 1 ;
   ```

   This severe problem has been corrected in MIL-STD-1589B.

2. Status types cannot conveniently be used in a well-structured programming style for loop iterations.

   EX : Given the type declaration

   ```
   TYPE st STATUS (V(STAT1) , ..., V(STATN)) ;
   ```

   and the loop statement

   ```
   FOR I : FIRST (st) THEN NEXT (I,1) WHILE (I≤LAST(st));
           loopbody;
   ```

   Because the THEN-effect evaluates NEXT(I,1) and sets I before the WHILE-effect checks for an exit condition, when I=LAST(st), NEXT(I,1) returns an illegal value for I, because it is outside the valid range of values for st.

Other areas with particularly complex semantics include table declarations, scope rules, and COMPOOLs.

There are many areas of JOVIAL(J73) in which users are likely to misinterpret the semantics of the language. For example, the introduction of strong typing into an already existing language prohibited many common programming constructs such as those described below:

1. Because of the strong type-checking rules, and the fact that a table-entry was always considered to be of type "table", it was not possible, in MIL-STD-1589A, to use bodiless tables as arrays, as was permitted in JOVIAL(J73/I). JOVIAL(J3B), which had strong typing, also had an ARRAY data structure, but this construct was not included in JOVIAL(J73) either. As a result, the only way to use an array in JOVIAL(J73) was to declare a table with a single-item body, a clumsy and unnatural solution.

   EX : Given the table declaration

   ```
   TABLE TT(10) S ;
   ```

   one could not write

   ```
   TT(3) = 1 ;
   ```

   This severe problem has been corrected in MIL-STD-1589B.

2. Status types cannot conveniently be used in a well-structured programming style for loop iterations.

   EX : Given the type declaration

   ```
   TYPE st STATUS (V(STAT1) , ..., V(STATN)) ;
   ```

   and the loop statement

   ```
   FOR I : FIRST (st) THEN NEXT (I,1) WHILE (I≤LAST(st));
           loopbody;
   ```

   Because the THEN-effect evaluates NEXT(I,1) and sets I before the WHILE-effect checks for an exit condition, when I=LAST(st), NEXT(I,1) returns an illegal value for I, because it is outside the valid range of values for st.

Other areas with particularly complex semantics include table declarations, scope rules, and COMPOOLs.

All of this means that the role of the Validator is extremely important. Compiler writers, like other users, are likely to misinterpret the semantics of the language and thus implement them erroneously, so it is essential that the Validator test the semantics thoroughly. We estimate that a truly comprehensive Validator would have to be in the neighborhood of 100,000 lines long.

THE JOVIAL(J73) COMPILER VALIDATOR

The acceptance and installation of compilers that perform poorly is commonplace. Because compilers are fundamental to the development of most software, the effects of attempting to use an unsatisfactory compiler are widely distributed. One reason for the existence of poor compilers is that the collections of programs used for testing them are themselves inadequate. There are several reasons for this:

1. Test set construction methods are not adequately systematic. In general, the tests reflect an ad hoc analysis of the language, the test designer's experience with other compilers (perhaps in different circumstances), and the competence of the people writing the test set.

2. Test set construction methods are not designed with reference to measures of test effectiveness. The lack of measurable, clearly defined objectives means that test construction is conducted in a fuzzy manner without guidance as to what should be tested or the degree of test thoroughness that ought to be sought.

3. The difficulty of constructing a good test set has usually been underestimated. Not until the validator is actually constructed do the intricacies of the language become apparent.

4. The necessity for constructing a good test set has usually been underestimated. The customer is often so eager to accept a compiler that, despite past experience, he assumes that it is basically correct and that only a small amount of acceptance testing is required. Hence, insufficient resources (time and money) are allocated for the validation development.

TRW had originally planned to use the Test Effectiveness Measurement Facility and a SEMANOL(76) specification of JOVIAL(J73) to guide construction of its JOVIAL(J73) Validator. However, due to a number of circumstances detailed elsewhere in this report, the Validator was constructed manually. Nevertheless, because of a great deal of research into what constitutes a good validator and a great deal of thought about organizational issues, we believe we have developed a superior product.

The remainder of this section describes the philosophy behind the Validator, its organization, its contents, and its status. Additional details on the operation of the Validator can be found in the Test Set Users Manual.

Philosophy

To avoid the pitfalls of prior validation efforts, it was deemed necessary to develop some goals for the Validator. First, it was decided that the Validator must have a uniform structure. Once this structure is developed, additional tests can be added, as necessary, without disturbing the established framework. The Validator also had to be flexible. This flexibility has two forms. It had to be possible to run the entire Validator or, alternatively, any subset of it. The structure adopted enables this flexibility. In addition, the Validator had to be designed to run on many computers; as a result, all machine parameters had to be identified. Our Validator is highly parameterized to isolate these machine dependencies. Unfortunately, flexibility and generality are always achieved at some cost - in this case, ease of use. In particular, the Validator cannot simply be run; rather, test suites must actually be constructed from the tests provided.

A second goal for the Validator was that it be highly informative. In particular, test results should be easily analyzed by the user. To this end, tests are self-checking wherever possible, results are concise and consistently formatted, and the probable source of a test failure is usually apparent. In order to provide these capabilities, a relatively complex output package had to be developed. This package is approximately 1500 lines long, and contains 39 JOVIAL(J73) procedures as well as one FORTRAN routine. It uses many complex features of the JOVIAL(J73) language and can be used as a benchmark test for the compiler. It must be installed before any of the test cases can be executed; however, all test cases can be compiled without it.

The Validator was constructed in the spirit of adversary conflict. That is, tests were not constructed with the assumption that the compiler is correct, but rather they were designed with the aim of discovering the flaws of the compiler. One of the consequences of such an attitude is that tests can depend on language features only to the extent they have been previously validated. As a result, tests were constructed using a small subset of the language. Some features in this subset are used for obvious reasons, while for others the justification is not as clear. Many features in the latter category are used only in a limited number of areas where their use was deemed essential. Except for the output package, the Validator uses only the following features:

1. Item Declarations;
2. Constant Item Declarations: to parameterize tests;
3. Integer Arithmetic;
4. Assignment;
5. IF Statements;
6. Relational Expressions;
7. Procedure Declarations and Calls;
8. COMPOOLs: to import the output package and parameterize tests;
9. DEFINEs: to enhance readability and parameterize tests;
10. Types: to parameterize tests;
11. LOC and Dereferencing: to look at machine representations of data, check parameter passing mechanisms, provide an alternate path to a variable;
12. BITSIZE, BYTESIZE, WORDSIZE, REP, Bit Conversions: to look at machine representations of data.

Of these features, the only one used extensively that is not relatively simple to implement is COMPOOL. Its use is justified for a number of reasons. First, it is machine independent. Machine parameters and DEFINEs could alternatively be imported using the !COPY directive, but its format is machine dependent; thus, every test program would have to be modified when rehosting the Validator. In addition, the traditional method of testing the COMPOOL feature has been to ignore it (the original JOVIAL(J73/I) JCVS had no tests for COMPOOL) or, at the very least, to vastly underestimate its testing requirements. This has caused severe problems, because COMPOOL is a very heavily used feature, and one whose correct operation is basic to program execution. Therefore, in this project, we took the opposite point of view; namely, that it is essential to test COMPOOLs first. Furthermore, it makes sense to test COMPOOLs in a production setting, such as the Validator itself, rather than test isolated components, since it is precisely in the combination of features that errors tend to occur. Thus, COMPOOLs have been made an inherent part of the Validator; the Validator will not run unless COMPOOLs have been properly implemented.

After a careful analysis of JOVIAL(J73) and extensive research into validation techniques, it became clear that it would not be possible to construct a "complete" validator with the resources available. Furthermore, it was not at all clear that, given the resources, one would want to construct such a validator; it would be at least 100,000 lines long. Priorities thus had to be adopted. It was decided that Validator coverage should be broad rather than deep. That is, it was important that all features of the language be covered to some degree, even though some areas could not be covered to the desired depth. Beyond that, extra attention was given to some particularly complex areas such as item declarations and table declarations, as well as to areas whose semantics were particularly well-specified, such as statements and intrinsic function calls.

Organization

The Validator has five components:

- A compool, SYNTAX, which supports the notation of the Validator;
- Two compools, TARGET and FLAVOR, used to parameterize the Validator for a particular compiler implementation;
- A machine independent output package which provides the Validator with the report capabilities it requires;
- A compool, MASTER, which insulates the Validator from local file name conventions;
- Test cases.

The SYNTAX compool contains a standard set of DEFINE declarations to enhance the readability of the Validator, as well as additional DEFINE declarations which implement useful compile-time functions.

The Validator contains two implementation-dependent compools, TARGET and FLAVOR. TARGET contains all the installation-dependent parameters which, once set, are unlikely to change during the course of a validation. These include the values of the set of JOVIAL(J73) implementation parameters, and also a number of descriptors and formatting parameters required by the output package. FLAVOR contains parameters which may be changed several times during the course of a validation. These ensure that data items with different sizes (and thus, presumably, different machine representations)

are used by the Validator.

The output package is written entirely in JOVIAL(J73) with the exception of a single seven line FORTRAN subroutine. Installations which do not have a FORTRAN compiler must recode this subroutine in some locally available language. The output package provides character stream output for the data types signed and unsigned integer, float, fixed, bit, and character. It also allows conversion between a variety of data types, and performs a few minor utility functions.

The compool MASTER consists of a single declaration for the define name IMPORT'VALIDATOR'SUPPORT. The definition part consists of COMPOOL directives to import the six support compools used by the Validator: SYNTAX, TARGET, CHANGE, GLYPHS, USEFUL, and SCRIBE. If the Validator is to run on a machine that does not accept file names in the above format, only MASTER needs to be modified.

The major portion of the Validator is composed of JOVIAL(J73) test programs arranged according to MIL-STD-1589A section number. Each section occupies a separate file. Each test within a section is assigned a structural form, a class, a time at which results are delivered, and a sequence number.

Every test has one of five structural forms: COMPOOL, PROCEDURE, PROGRAM, ROUTINE, or MIXED. The first three forms are compool, procedure, and main program modules, respectively, as defined in MIL-STD-1589A. Tests designated as ROUTINEs are parameterless procedures which may be combined into procedure or main program modules as the user sees fit. Tests of the form MIXED will contain a variety of subunits; one such test might contain a compool module, a main program module, and two procedure modules. It is the user's responsibility to extract these subunits for submission to the compiler.

Tests also belong to one of five different classes: CONFORMANCE, DEVIANCE, IMPLEMENTATION, QUALITY, and CAPACITY. CONFORMANCE tests are always legal JOVIAL(J73) and should compile without error if the compiler is standard conforming, although warnings may be issued by the compiler. These tests are derived directly from MIL-STD-1589A and attempt to ensure that processors provide the features mandated by the standard and behave as required. Tests of class DEVIANCE are never legal JOVIAL(J73), but differ from it in some

subtle way. They serve to detect processors which fail to adhere to a stipulated constraint of the language standard, incorporate some common error, or fail to check or limit some JOVIAL(J73) feature appropriately. Tests in this class verify that the compiler being tested rejects language which is not legal JOVIAL(J73). The class IMPLEMENTATION explores those portions of the language standard where implementers are permitted some freedom in implementing a feature; in many cases MIL-STD-1589A flags these features with the phrase "implementation-dependent." The class QUALITY consists of tests which have as their only common feature that they explore in some sense the quality of an implementation. These tests include benchmarks whose space and execution time can be measured to estimate performance. Finally, the class CAPACITY explores the compiler's ability to handle unusually large or deeply nested constructions.

Each test is expected to deliver its results at a particular time. COMPILE time tests are evaluated by examining the compiler listings. OBJECT time tests are evaluated by examining the object code generated by the compilation of the test. RUN time tests are evaluated by studying their behavior during execution. Tests whose time is OUTPUT produce printed reports containing the result of the test.

Every file of tests, corresponding to one section of MIL-STD-1589A, is logically subdivided into suites. A suite is a group of tests, each of which has the same form, class, and result time. In theory, a file may contain up to 100 suites, since there are five possible forms, five distinct classes, and four different result times, but in practice no file contains more than five.

Suites within a file are arranged in increasing order according to a three key sort where the primary key is the form, the secondary key is the class, and the tertiary key is the result time. The sort order is:

Form: COMPOOL < PROCEDURE < PROGRAM < ROUTINE < MIXED

Class: CONFORMANCE < DEVIANCE < IMPLEMENTATION < QUALITY < CAPACITY

Result: COMPILE < OBJECT < RUN < OUTPUT.

Within a suite, tests are further ordered by paragraph number within the pertinent section of MIL-STD-1589A and within a paragraph, tests are ordered, where possible, by sentence reference.

Each suite begins and ends with comments whose form is fixed and consistent throughout the Validator. The suite header is composed from the section number, form, class, and result, and has the following form:

%SECTION = section-number FORM = form CLASS = class RESULT = result%

where section-number is derived from MIL-STD-1589A, and form, class, and result have values as given earlier. The suite trailer comment has the form

%END SECTION section-number%

For the convenience of those users who lack sophisticated text editing and extraction tools, the suite header is also encoded as a fixed format comment called a key. To build a key, each form, class, and result time is assigned a numeric value. The forms COMPOOL, PROCEDURE, PROGRAM, ROUTINE, and MIXED are assigned values 0 through 4, respectively, as are the classes CONFORMANCE, DEVIANCE, IMPLEMENTATION, QUALITY, and CAPACITY. The result times COMPILE, OBJECT, RUN and OUTPUT are assigned the values 0 through 3, respectively. The key comment then has the form

%KEY section-number' sequence-number form' class' result'%

where section-number' is derived from section-number, sequence-number is a three-digit number, and form', class', and result' are the numeric encodings of form, class, and result, respectively. Keys which follow suite headers always have the sequence-number 000.

CAPACITY tests are handled somewhat differently. Their bulk and the difficulty of relating a given capacity test to a specific section of MIL-STD-1589A require that each capacity test be placed in a separate file. For these tests, each file contains precisely one suite and that suite contains only one test. In addition, to remain consistent with the conventions of the Validator, capacity tests are assigned the fictitious section number 99.0. It is unlikely that this imaginary section number will conflict with later versions of language standard, since MIL-STD-1589A contains only 9 chapters.

Each test within the Validator follows a consistent framework. Each test in a section is assigned a three digit sequence number which is unique to a section but not necessarily across sections. The first test in a section is assigned the sequence number 001; succeeding tests are assigned sequence

numbers in increments of one. Note that the leading zeros are required.

Each test is bracketed by header and trailer comments whose form is fixed and consistent throughout the Validator. A test header comment has the following form:

%TEST = test-number FORM = form CLASS = class RESULT = result%

where test-number is composed of the section-number and sequence-number separated by a dash, and form, class and result are as given for suite header comments. The test trailer comment has the form

%END TEST test-number%

Like suite headers, test headers are followed by a key. Keys which follow test headers contain the sequence number given in the test header.

Each test name is a uniform encoding of the test's section number, sequence-number, form, class, and result time. The prefix of the test name is taken from the form, as follows:

C denotes COMPOOL module
P denotes PROCEDURE module
M denotes MAIN PROGRAM module
R denotes ROUTINE.

Sub-component names for the form MIXED are generated in a test dependent manner. The suffix of the test name is constructed from the test number, form, class, and result by replacing "." with "P", "-" with "D" and appending the numeric encodings of the test form, class, and result time. Note that test names are unique across the validator, that is, no two tests ever have the same name.

Immediately following the test header and key is an explanatory comment whose arrangement depends on the class of the test. If the test is of class CONFORMANCE, IMPLEMENTATION, or QUALITY, the remark will have three components: a one sentence summary of the test, a detailed explanation of the precise issue under test, and a list of references to MIL-STD-1589A, arranged in increasing order in the obvious fashion. For tests of class DEVIANCE the leading remark has only two components: an explanation of the precise issue under test, and a list of references to MIL-STD-1589A. For CAPACITY tests

the leading remark only contains an explanation of the issue under test.

Tests whose result time is OUTPUT are terminated by a comment which outlines the printer report the user can expect to see if the test executes successfully. This comment has the form

```
%SAMPLE test-number
   :
   :
 test dependent text
   :
   :
END SAMPLE test-number%
```

The internal test structure is dependent upon the test, class, and result time of the test. In what follows, if a form, class, or result time is not explicitly mentioned, then no standards apply.

If the form is PROCEDURE, PROGRAM, or ROUTINE, then all declarations at the outermost level of scoping of the test are bracketed by declaration header and trailer comments of the form

```
%DECLARATIONS test-number%
%END DECLARATIONS test-number%
```

These brackets are provided to allow the easy, semi-automatic construction of a wide variety of compool tests.

If the class is DEVIANCE, then all illegal constructions are flagged with the JOVIAL(J73) comment % ILLEGAL. %

If the test class is CONFORMANCE, DEVIANCE, IMPLEMENTATION, or QUALITY, and the result time is RUN or OUTPUT, the test will have the general form

```
declarations
report headings
{initialization}
subtest #1
   :
   :
{initialization}
subtest #n
evaluation
report trailer
```

The report headings always begin with the comment

```
% REPORT HEADINGS. %
```

which is immediately followed by the JOVIAL(J73) statement

```
WRITES ('!NTEST = test-number CLASS = class RESULT = result !N');
```

(WRITES is a routine provided with the output package.) The report headings will usually contain other test dependent calls to the output package as well.

Each test consists of one or more subtests and each subtest can be preceded by an optional initialization section. Each initialization section begins with the comment

```
% INITIALIZATION. %
```

and each subtest begins with the comment

```
% TEST number. %
```

where number is the ordinal number of the subtest. Each subtest usually consists of only a few lines of code - few subsets are more than 10 lines long.

The evaluation portion of the test is always preceded by the comment

```
% EVALUATION. %
```

This component of the test determines the final outcome of the test based on the results of the previous subtests.

The report trailer is always the single statement

```
WRITES('END test-number !N');
```

it is always the last executable statement of the test.

Many tests contain pre- or post-conditions encapsulated as comments in the form

```
% ASSERT ... condition text ... . %
```

Assertions are intended as guidelines for users modifying tests to suit local installation-independent needs. Assertions are usually, but not always, associated with DEFINE, CONSTANT, or TYPE declarations and state conditions which must hold if the test is to run correctly.

Contents

The Validator contains approximately 50,000 lines of JOVIAL(J73) code, not including capacity tests or the output package. Some tests exist for almost all sections of MIL-STD-1589A, but some areas are covered to a much greater degree than others. In particular, the Validator tests the following sections especially well:

- 1.4 Implementation Parameters
- 2.1.1.1 - 2.1.1.7 Item Declarations
- 2.1.2.1 - 2.1.2.4 Table Declarations
- 4.1.1 - 4.1.9 Statements (except for ABORT)
- 6.3.1 - 6.3.9, 6.3.11 Function Calls (except for NWDSEN)

A few sections were not tested because their syntax is implementation-depdendent. These include:

- 3.5 Machine-Specific Procedures and Functions
- 9.2.1 Copy Directive
- 9.3 Linkage Directive
- 9.8 Register Directive

In addition, the following sections were not tested at all because of time constraints:

- 3.4 Inline procedures and functions
- 4.10 ABORT statement
- 6.3.10 NWDSEN function
- 8.5 Blanks
- 9.2.2 SKIP, BEGIN, END directives
- 9.4 TRACE directives

It should be noted that several other sections appear not to be tested (or tested only superficially), but are actually implicitly tested in the test cases for other sections. A list of these sections follows, together with the section where they are actually tested:

- 1.2.2 Procedure modules (production tested in building test suites)
- 1.2.3 Main program modules (production tested in building test suites)
- 2.7 Null declarations (distributed)
- 4.5 Procedure call statements (3.1, 3.3)
- 5.4 Status formulas (2.1.1.6)
- 5.5 Pointer formulas (2.1.1.7)
- 5.6 Table formulas (2.1.2, 2.1.2.1, 2.1.2.2, 2.1.2.3, 2.1.2.4)
- 9.11 Allocation order directive (2.1.2.3)

Several other sections were not tested to the degree desired. These are listed below, together with a description of the deficiency:

- 1.2.1 COMPOOL Modules
  9.1 COMPOOL Directive

  Although production tests for these sections are provided via the output package, no tests of specific properties are provided. Such tests would address such issues as scoping, variable attributes, structures, presets, REF/DEF, interaction with other directives, and the form of the COMPOOL directive.

- 1.3 Scope of Names

  The standard is quite vague on this subject and additional testing would be desirable.

- 1.4 Implementation Parameters

  Although, in general, this section is well-tested, it would be desirable to provide deviance tests which input a parameter of the wrong type to those parameters which are actually functions.

- 2.2 Type Declarations

  No tests exist for the LIKE option, or for matching of table or block types.

- 2.5 External Declarations
  2.5.1 DEF Specifications
  2.5.2 REF Specifications

  The tests required for these sections are similar to those for COMPOOLs.

- 2.6 Overlay Declations

  No tests exist for chained overlays, overlays within a block, overlaying an object on itself, or bizarre overlay constructs.

- 5.1.1 Integer Formulas

  No tests exist for the MOD or ** operators.

- 5.1.2 Floating Formulas

  No tests exist for the ** operator.

- 6.1 Variables

  No tests exist for the BIT, BYTE, or REP pseudovariables, for variable references inside blocks or tables, or for pointer dereferences.

- 6.2 Named Constants

  More tests of the semantics of this section should be written.

- 7.0 Type Matching and Conversions

  No tests exist for table conversions, and few address both compile-time and run-time conversions.

It would also be desirable to construct data-dependent tests for some sections, as errors frequently appear only when features interact with one another. For example, it would be useful to construct tables or overlays which contain several different data types.

Fourteen capacity tests were constructed. These check all the capacity requirements from the JOCIT/J73 Compilers Part I Product Specification except that the compiler can efficiently translate:

- a program consisting of not less than 150,000 essential characters;
- a program generating as much as 256,000 words of machine code and data;
- a program referencing !COPY directives with at least 10 levels of nesting;
- a procedure call with at least 20 parameters.

The Validator is weak in the testing of compiler error messages, in particular, it contains virtually no tests which deliberately exercise context-free syntax errors. It was originally planned that such tests be based on the syntax errors discovered in the process of testing the Validator itself, thereby rooting these tests in the beginning of a representative sample. This process was, however, never implemented, because the Validator tests were never actually compiled. Nevertheless, it should be noted that, with a few exceptions, all constraints explicitly identified in MIL-STD-1589A have been tested.

Although it may appear that a lot is missing from the Validator, it should be emphasized that the Validator is already 50,000 lines long. If all the "holes" that have been identified were filled, we estimate that the Validator would be in the neighborhood of 100,000 lines long.

Testing

It was planned that the Validator test cases be debugged using a GFE JOVIAL(J73) compiler. A subset compiler was to be delivered in October, 1979 and a production-quality compiler in January, 1980. A JOVIAL(J73) compiler, was in fact, delivered to TRW in February, 1980, and improved versions of that compiler were delivered in March, April, July, September, and October, 1980, as well as in January, 1981. None of these compilers is production-quality, with the possible exception of the last; by the time it arrived, there was virtually no time remaining in the contract for test case development. In fact, only a few compools have been successfully compiled. Thus, although the test cases have been proofread extensively, the Validator is, in no sense, debugged.

THE SEMANOL SPECIFICATION OF JOVIAL(J73)

SEMANOL (an acronym for SEMANtics Oriented Language) is a unique TRW system for formally describing the syntax and semantics of contemporary programming languages. A SEMANOL specification achieves a clarity and precision that prose specifications lack; thus, compiler writers and other users should find the SEMANOL interpretation helpful in understanding complex languages like JOVIAL(J73).

A SEMANOL specification can play several roles in the testing process as well. Most importantly, its use as a source document for test case derivation will result in better quality tests being produced, because test case formulation can be done more systematically than otherwise. The SEMANOL specification also provides a means by which the test set can itself be tested; this is particularly useful when no compilers for the language being tested yet exist. The test programs can be processed by the SEMANOL Interpreter, under control of the SEMANOL specification, and so can be validated against a uniformly understood specification. Finally, the SEMANOL specification provides a basis upon which objective measures of test set effectiveness can be computed. The quality of a test set can then be measured relative to a given standard, for example in terms of SEMANOL definitions exercised or execution paths taken through the SEMANOL specification.

The SEMANOL specification of JOVIAL(J73) that was developed under this contract was used primarily as a reference document. Despite its limited area of application, the specification was extremely useful in identifying possible implementation models (both correct and incorrect), difficult-to-implement areas of the JOVIAL(J73) language, and problems left unresolved by the less formal language standard. For example, the design of the parse phase of the SEMANOL specification required a detailed study of the semantics of DEFINE, which revealed unexpected interactions with the equivalence of uppercase and lowercase letters in program text. This suggested that the tests for a compiler's implementation of DEFINE ought to include tests which exercise those interactions.

It should be noted that the JOVIAL(J73) test cases developed in this contract were not processed by the SEMANOL Interpreter. Likewise, they were not input to the Measurement Facility. To have done so would have required enormous computer resources (both time and space) which were not available to this project. Nevertheless, there is no inherent reason that this could not be done at some later time.

An introduction to the SEMANOL(76) metalanguage, the most recent dialect of SEMANOL, is presented next, followed by a discussion of the changes to the language made for this project. Complete documentation of the SEMANOL specification of JOVIAL(J73) itself appears in Appendix A of this report, while documentation of the modified SEMANOL(76) Interpreter is contained in Appendix C.

An Introduction To SEMANOL(76)

SEMANOL(76) is intended for use in describing (procedural) programming languages. A specification written in the SEMANOL(76) metalanguage is meant to provide an exact and complete definition of a programming language that is comprehensible to a suitably trained reader. That is, SEMANOL(76) is designed to supply people with a basis for communication about programming languages that is more precise than commonly employed description methods. Additionally, specifications written in SEMANOL(76) are executable.

The specification method adopted is algorithmic. This choice stems from a feeling that the semantics of programming languages ought to be explained in this way. That is, semantics are concerned with explaining how something happens and not just in characterizing an input-output relationship. Certainly this is the way in which designers, compiler writers, and application programmers generally view the semantics of a programming language. Having a direct correspondence between the formal, operational, SEMANOL(76) expression of language semantics and a reader's intuitive conception of a language yields a specification method that can be easily understood. An algorithmic method also permits language details, such as those specific to a given implementation, to be described exactly when desired.

The SEMANOL(76) method considers a programming system, S, to be defined by $S = (P, I, T, \Phi)$ where

- P = The set of programs which can be expressed in the programming system.
- I = The set of input values.
- T = The set of output traces. The trace is an ordered record of significant actions (such as assignment) that are performed by the program as it is executed; it is the visible manifestation of performing the algorithm that is the operational SEMANOL(76) specification of semantics. The trace is thus similar to a state sequence.
- Φ = The semantic operator. This operator, given as $\Phi : P \times I \rightarrow T$, is considered to define the "meaning" of a program.

P, I, and T are each sets of strings which are specified by Φ, and whose individual members will be denoted by the corresponding lower case letters (i.e., $p \in P$, $i \in I$, $t \in T$.). The effect of executing a given program, p, can then be denoted in terms of the semantic operator by

$$\Phi(p,i) = t$$

Thus Φ specifies the trace produced by any program in the system when that program is executed with any input value sequence. The SEMANOL(76) metalanguage is used for programming the semantic operator, thereby providing a method for formal specification of a programming language. Since the SEMANOL(76) metalanguage is itself a programming language, it also belongs to a programming system. To differentiate between these two systems, we will use the subscript j to identify elements of the programming system being defined by a SEMANOL(76) program (e.g., JOVIAL(J73)) and the subscript s to identify elements of the SEMANOL(76) system. The semantics of a JOVIAL(J73) program p_j, are then expressed by

$$\Phi_j(p_j,i_j) = t_j$$

Note that, since JOVIAL(J73) has no facilities for input, i_j is always null.

The semantic operator for JOVIAL(J73), Φ_j, is expressed as a SEMANOL(76) program, p_s, which in turn is interpreted by a semantic operator for the SEMANOL(76) programming system, Φ_s.

Thus we have

$$\Phi_s(p_s,(p_j,i_j)) = \Phi_j(p_j,i_j) = t_j$$

and a formal definition of JOVIAL(J73) is provided by p_s. The definition of the SEMANOL(76) semantic operator, Φ_s, is given by the SEMANOL(76) Reference Manual and has been implemented by the SEMANOL(76) Interpreter computer program.

This general view of language definition is shown graphically in Figure 1. As shown there, these levels of semantic specification correspond to defining a virtual machine for SEMANOL(76) and, based on that, one for JOVIAL(J73).

As observed, the SEMANOL(76) metalanguage is meant to describe semantic operators. Consequently, it is a high level language designed for the specific purpose of completely and exactly describing the syntax and semantics of procedural programming languages. The metalanguage permits high-level expressiveness and makes no special effort to minimize the primitives available. It thus contains some redundancy in its primitives and permits syntactic variations where this can aid reader interpretation. Where possible, "conventional" notation, as found in mathematical exposition and in other programming languages, is employed so that a reader's intuition will generally lead to a correct interpretation of SEMANOL(76) code. The semantics of execution are described by the use of SEMANOL(76) in terms of parse trees and elements of the original source program text, and so can be directly understood by the reader.

The SEMANOL(76) metalanguage has many features common to other programming languages, such as imperative, conditional, and iteration control statements; Boolean constants and functions; procedure definitions; recursion abilities; a rich set of character string operators; and functional definitions that provide case selection. However, its unique domain of application means that many of its features are not so common. SEMANOL(76) provides facilities for defining a context-free grammar, including a feature for context-sensitive specification of where spaces may appear, and couples that with an operator for generating a parse tree for a given string from that grammar. Various operators are then available for use upon this parse tree, including a group for tree traversal. SEMANOL(76) deals with sequences and offers high-level iterators, including existence tests, for use on these structures. Arithmetic is done on numeric

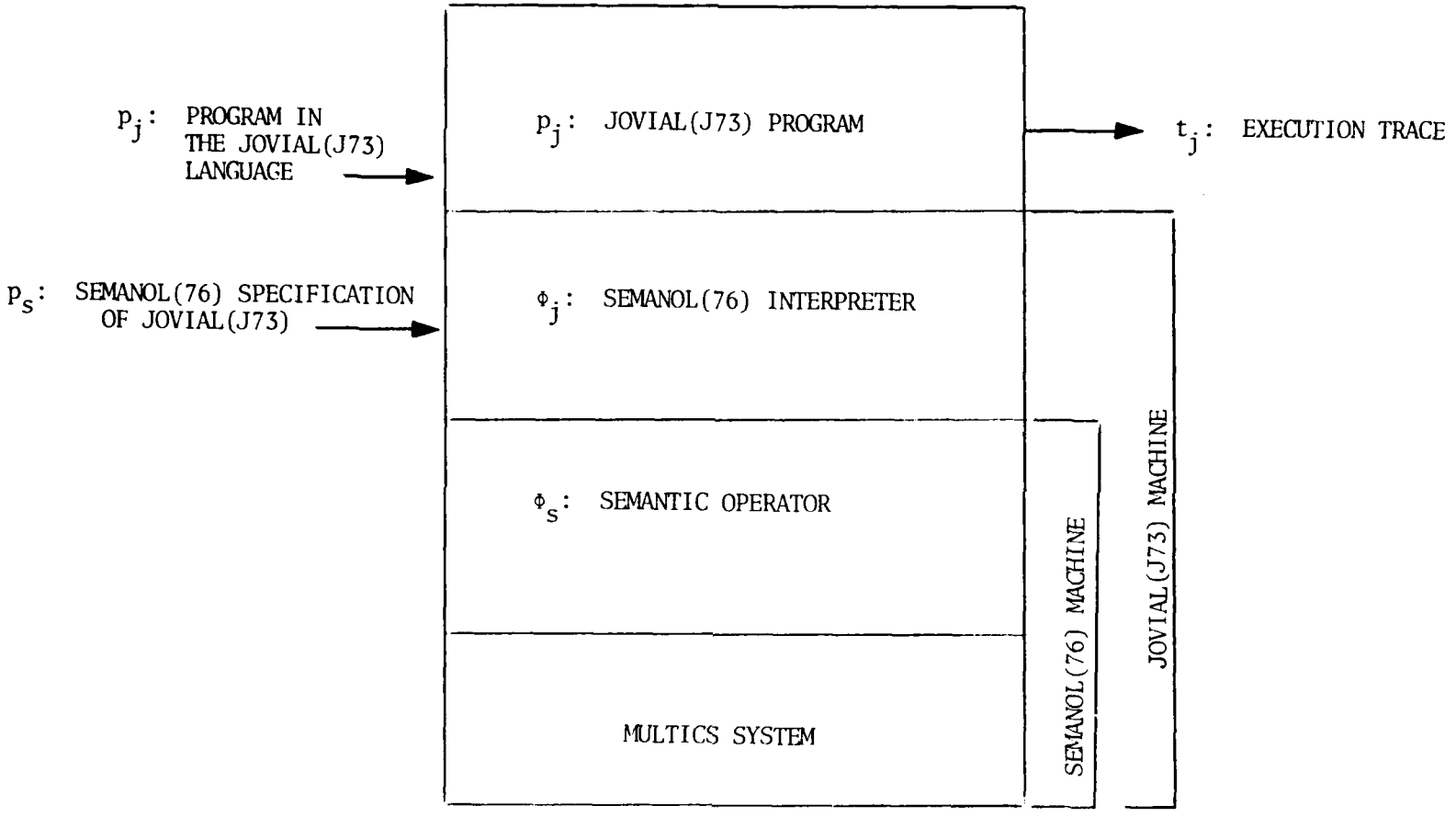


Figure 1: The SEMANOL(76) System

strings and so has a significance that is independent of host machine factors; the arithmetic specified for JOVIAL(J73) is controlled by the one doing the specification in SEMANOL(76) and not dictated by the fact that the SEMANOL(76) system is implemented on a HIS-6180 computer. Assignment and reference are accomplished through use of a single level associative storage mechanism. An effort has been made in designing the SEMANOL(76) metalanguage to provide these features in a manner that would stress readability; it is a language where the prospective reader's viewpoint has been a dominant influence.

Programs written in SEMANOL(76), such as the specification of JOVIAL(J73), can then be processed by the SEMANOL(76) Interpreter. The SEMANOL(76) Interpreter accepts a SEMANOL(76) specification of a programming language and uses that input specification to realize the semantic effect of (i.e., to execute) programs written in the language thus defined. By virtue of the Interpreter, SEMANOL(76) specifications can themselves be tested and debugged. Furthermore, an operational standard for the defined language is thus created.

The operation of the elements that constitute the SEMANOL(76) system is shown in Figure 2. The broken line encloses the SEMANOL(76) Interpreter, which can be seen to actually consist of two loosely connected programs identified as the Translator and the Executer. The Translator accepts the SEMANOL(76) program describing a programming language and converts it to SIL code. The SIL code is an alphanumeric representation that is much more conveniently processed than the original text. The SIL file is read by the Executer program, and the SIL code is then used to control, or drive, the Executer program. The present Interpreter is operational upon the HIS-6180 Multics System.

It is to be emphasized that this system is interpretive, and that neither the defined language program, nor the SEMANOL(76) program describing the defined language, are translated (i.e., compiled) to machine code. The JOVIAL(J73) program text is interpretively "executed" by the SEMANOL(76) program describing the JOVIAL(J73) language, while the SEMANOL(76) program text (i.e., the SIL code) is, in turn, interpreted by the Executer program. This two-level interpretation does mean that the execution time of test programs is slow. In fact, the situation is acceptable only for very small test programs.

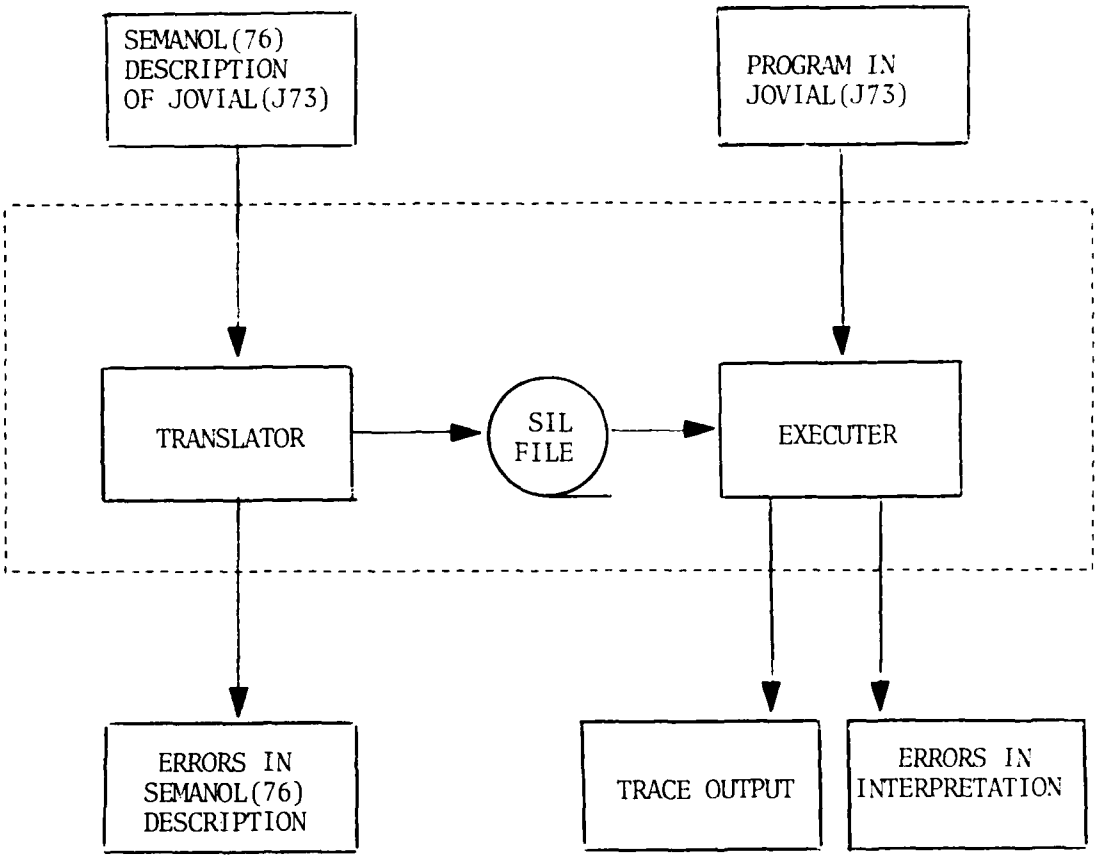


Figure 2: SEMANOL(76) Interpreter Logic

Modifications to SEMANOL(76)

SEMANOL(73), the predecessor language to SEMANOL(76), had a construct, #RESUME(;n), which allowed the semantics of preemptive evaluation to be conveniently modelled. This feature was used to specify the control semantics of JOVIAL(J73/I) in the SEMANOL specification done for that language. #RESUME, unfortunately, also had a number of disadvantages. The most serious of these was that its semantics were extremely complicated and likely to be misunderstood, and were also likely to be incorrectly implemented. Since its use was not deemed necessary in the SEMANOL specifications of Minimal BASIC, JOVIAL(J3), and Ada, #RESUME was not included in SEMANOL(76).

Nevertheless, the eliminated function was deemed to simplify the specification of JOVIAL(J73) control semantics, particularly ABORT statements and GOTO label parameters. Without such a capability to preempt partial and pending DF-evaluations, the SEMANOL specification would have two alternatives for the treatment of abnormal terminations in JOVIAL(J73). It could guard the contexts of all evaluations of the value-of DF to achieve the possible preemptive effects of abnormal terminations of function calls. Alternatively, it could designate each formula evaluation as an executable step and store temporary values of formulas in a dynamic stack. Either of these alternatives would complicate further the already complex evaluation semantics. Instead, a new capability has been implemented which allows the SEMANOL specification of JOVIAL(J73) to confine the description of abnormal routine termination to the control-semantics definitions that are specifically concerned with such termination. The remainder of this section describes the role of the new keywords #TRY and #FAIL-WITH-VALUE in the evaluation of expressions; this discussion also appears in the revised SEMANOL(76) Reference Manual produced as an adjunct to this project.

In normal evaluation of a SEMANOL(76) expression, e, its subexpressions will be evaluated, as will any semantic definitions invoked in those evaluations; thus a long dynamic chain of (sub) evaluations may take place as part of evaluating e. At any point in the chain of evaluations, some will have completed, leaving pending results; others will remain to occur, and will use those pending results. The evaluation of e itself will often be part of a

larger chain of evaluations contributing to the evaluation of some outer expression. The evaluation of such an outer expression will begin before its sub-evaluations begin, and end after they end.

SEMANOL permits incomplete evaluation of expressions, using the coordinated notions of guards (indicated by the keyword #TRY) and failure-events (indicated by the keyword #FAIL-WITH-VALUE). Complete evaluation of #TRY el consists of

(1) creation of a guard for el
(2) evaluation of el, and
(3) deletion of the guard for el.

The guard is used only to handle (or "field") failure-events which occur during the evaluation of el. If all three steps are completed normally, the value of the expression #TRY el is just the value of el and the execution/evaluation process continues in normal sequence.

As part of the evaluation of el, evaluation of an expression #FAIL-WITH-VALUE vl may occur; this results in a failure-event which is fielded by the most recently created guard. If the failure-event is fielded by the guard for el, then the evaluation of el is interrupted and fails to complete. All intermediate evaluation results obtained to that point in the evaluation of el are lost. (Assignments performed during the evaluation are not reversed, however, and there may thus be a residual state change - a side effect.) At this point, the evaluation of el is replaced by the evaluation of vl. When evaluation of vl is completed, evaluation continues at step (3), deletion of the guard for el.

It may happen that unfielded failure events occur in the evaluation of vl, thus causing its evaluation to be replaced with that of, say, v2. Such replacement may occur to an arbitrary depth. If there is no guard to field a failure-event, evaluation terminates with value <error>.

THE TEST EFFECTIVENESS MEASUREMENT FACILITY

The idea of measuring the quality of a compiler validation test set by examining its coverage of a SEMANOL(76) specification of a programming language was first studied several years ago while performing Contract F30602-76-C-0255, described earlier. The premise is that, since a SEMANOL(76) specification is itself a program, it is reasonable to consider applying the same test effectiveness measures to it that are proposed for programs generally. In particular, the idea of measuring test effectiveness in terms of coverage of structural elements was evaluated in relation to SEMANOL(76).

For syntactic #DFs, each production could be considered a distinct structural element; alternatively, each case of each production could be used as the basis for measurement. Several options were studied for semantic definitions. One method considered semantic #DFs and #PROC-DFs to be structural elements and measured effectiveness in terms of element coverage. A refinement of this method would use a finer definition of structural elements so that each case of a semantic #DF and each statement of a #PROC-DF would be defined as a structural element. An even more sophisticated technique measures test effectiveness by the proportion of control options taken at each branch point by a given input set. These methods all appear to provide useful measures, of increasing validity, of test effectiveness. They are easily understood, because they are stated in terms of the SEMANOL(76) specification itself. Furthermore, they can be effectively incorporated into iterative procedures for constructing and refining validators that are guided by test effectiveness results at each step.

During the design of the Test Effectiveness Measurement Facility (Measurement Facility, for short) implemented in this contract, several desired attributes were identified. First, it was to be based upon existing SEMANOL(76) tools, that is, the Interpreter system. In addition, because the Measurement Facility was no longer the primary focus of activity, it was important that it be kept as simple as possible. Therefore, the decision was made to base coverage on syntactic and semantic #DFs (and #PROC-DFs) rather than on a finer definition of structural elements. Another goal was to minimize the required changes to

the SEMANOL(76) Interpreter. An advantage of the effectiveness measure chosen is that it could be implemented without modifying the SEMANOL Intermediate Language (SIL) code produced by the Translator. Instead, the output trace from the Executer, augmented by the output from a parse tree tracer, is used as the main interface between the Executer and the Measurement Facility; only a one-line change to the Executer was required to effect this capability.

Despite the modest nature of the Measurement Facility, it has been designed in such a way that it is easily extendable to more refined measures of test case effectiveness, should that be desired. The major obstacle is associating a name with a structural element that does not already have a name; this must be done in the SIL code.

The Measurement Facility is not machine independent. It consists primarily of a collection of Multics ecs, or exec_com segments; these are essentially parameterized command sequences. Six of these ecs are called directly by the user; the remainder are called by other ecs. Two FORTRAN programs also are included in the Measurement Facility; these, too, are called by ecs. The Measurement Facility calls the SEMANOL Translator and Executer directly, of course, and also relies on several utility procedures developed in prior SEMANOL contracts.

The Test Effectiveness Measurement Facility has four major capabilities:

1. To prepare a SEMANOL(76) specification of a programming language for processing;
2. To prepare a suite of test cases for processing;
3. To add test results to a data base;
4. To inquire about test results in a data base.

Each of these four capabilities is discussed below. Additional information about the operation of the Test Effectiveness Measurement Facility can be obtained from the Test Effectiveness Measurement Facility Users Manual.

Preparing the SEMANOL(76) Specification

The SEMANOL(76) specification must be modified before it can be used by the Measurement Facility. In particular, a parse-tree tracer must be added and the distinction between actions performed during compilation and those performed during execution must be made explicit. These functions are performed by the ec create_spec_archive. This ec has three inputs. The first is the SEMANOL(76) specification itself. This file must have the name

semanol_program.object-language

For example, for this project, it would have the name

semanol_program.j73

as does our SEMANOL specification. The specification must be informally divided into a sequence of sections, each beginning with a header comment of the form

"#HEADER: <str1>,<str2>, <str3>,<str4>"

where <str1> is generally the language name, <str2> is the section name, and <str3> and <str4> are other text strings irrelevant to the Measurement Facility. In particular, the syntactic definitions must be preceded by a header comment with <str2> = syntax.

The second input to create_spec_archive is a language-independent segment named

syntax_trace.drivers.semanol

This file is a fragment of a SEMANOL(76) metaprogram provided with the Measurement Facility on the Multics file system, which, in conjunction with the file st.r.object-language.semanol (described below), enables the Measurement Facility to traverse the parse tree of a test program.

The final input to create_spec_archive is a language-dependent segment named

st.r.object-language.semanol

where object-language is the same as for the SEMANOL specification. This file must be provided by the user. It, too, is a fragment of SEMANOL(76) metaprogram code which, in conjunction with syntax_trace.drivers.semanol, enables the Measurement Facility to traverse the parse tree of a test program. It must have the following form:

```
#DECLARE-GLOBAL:
    global1, ... , globaln #.
```

```
#SEMANTIC-DEFINITIONS:
    {#DF      }
    {#PROC-DF}      trace-program-tree => ... #.
     #PROC-DF compilation ... #.
     #PROC-DF execution ... #.
```

The SEMANOL(76) variables globall, ..., globaln are global variables declared in the SEMANOL(76) metaprogram which are used in the semantic definitions trace-program-tree, compilation, and execution.

The semantic definition trace-program-tree is always required. For each parse tree, t, used to represent a test program, the definiens of trace-program-tree must invoke the #DF trace-tree (which is contained in the file syntax_trace.drivers.semanol). For example, if the test program is represented by a sequence of parse trees denoted by a global variable, say, seq-of-modules, as in JOVIAL(J73), st.r.j73.semanol would contain:

```
#DECLARE-GLOBAL : seq-of-modules #.
#SEMANTIC-DEFINITIONS:
     #PROC-DF trace-program-tree
          #FOR-ALL i #IN seq-of-modules
               #DO #COMPUTE! trace-tree (i) #.
```

The semantic definitions compilation and execution may be omitted if the Control-Commands section of the SEMANOL(76) metaprogram has the form

```
#CONTROL-COMMANDS:
     #COMPUTE! compilation
     #COMPUTE! execution #.
```

Otherwise, the actions corresponding to compilation from the original Control-Commands section must be incorporated into the #PROC-DF compilation. Likewise, execution actions from the original Control-Commands section must be incorporated into the #PROC-DF execution. The Measurement Facility will then replace the original Control-Commands section with one of the above form. Our SEMANOL specification of JOVIAL(J73) is already in the preferred format.

After execution of the create_spec_archive ec, a metaprogram archive will exist, containing the following segments:

1. An ASCII segment containing a copy of the original SEMANOL(76) specification.

2. An ASCII segment containing the SEMANOL Intermediate Language (SIL) version of (1).

3. An ASCII segment containing an index which lists all the SEMANOL(76) definition names used in (1).

4. An ASCII segment containing a fragment of SEMANOL(76) code which enables syntax measurement capabilities in the Test Effectiveness Measurement Facility. This segment contains the revised Control-Commands section, the contents of the input segments syntax_trace.drivers.semanol and str.object-language.semanol, and a new semantic definition, type(t), which allows nodes on the parse tree for a test program being measured to be listed.

5. An ASCII segment containing the SIL version of (1).

Preparing the Test Cases

Before a set of test cases can be input to the Measurement Facility, they must be collected into a Multics archive. This is accomplished by executing the archive_test_suite ec. This ec accepts a segment consisting of a sequence of test cases, and separates the test cases into distinct segments contained in a test suite archive. The test cases are required to be in the following format:

<begin-marker$_1$>TEST=α FORM=β CLASS=γ RESULT=$\delta$$_{\sqcup *}$<end-marker>

.
.
.
.

<begin-marker$_2$>END$_{\sqcup *}$<end-marker>

where:

α is a test number of the form

<digits> {.digits}* -<3-digit sequence number>

β is one of COMPOOL, PROCEDURE, PROGRAM, or MIXED

γ is one of CONFORMANCE, DEVIANCE, IMPLEMENTATION, QUALITY, OR CAPACITY

δ is one of COMPILE, OBJECT, RUN, or OUTPUT

<begin-marker>s and <end-marker> are chosen by the test set developer, but are normally comment delimiters for the language of the test set.

Any test data that is required for test execution must be in a similar format:

<begin-marker$_1$> DATA = $\alpha_{\sqcup *}$<end-marker>

.
.
.

<begin-marker$_2$> END$_{\sqcup *}$<end-marker>

Note that the JOVIAL(J73) Validator Test cases conform to this format. Of course, since JOVIAL(J73) has no facilities for input, the Validator has no files of the second type.

After execution of the archive.test.suite ec, a test suite archive will exist, containing one segment for each test case or data file. The test case names are of the form $\alpha'.\ \beta'.\gamma'.\delta'$ where

α' = α, preceded by a 'p', with each period in α replaced by a 'p', and the dash replaced by a 'd'.

EX: If α = 5.1.1-007, then α' = p5p1p1d007

β' = first four letters of β

γ' = first four letters of γ

δ' = first three letters of δ

Data segment names are of the form α'.data

It is possible to add test cases to an already existing test suite archive. This function is performed by the augment_archived_test_suite ec. This ec requires, as input, the file name of the archive and the name of the file which contains the new test cases. The latter file must be in the format described above. The ec prune_archived_test_suite allows test cases to be deleted from the archive. It requires the file name of the archive and the name of a file which contains the names of the tests to be pruned.

Adding Test Results to the Result Data Base

After the metaprogram archive and test suite archive have been created, they can be used to add test results to the result data base. This is done with the

monitor_tests ec. This ec inputs test programs (and, optionally, test data) from a test suite archive to the SEMANOL(76) Interpreter using a specified metaprogram archive. All of the test programs in the specified test suite archive may be processed, or only a subset of these tests, called a Test Interest Group (TIG).

After a test is executed by the SEMANOL(76) Interpreter, monitor_tests processes the trace output. In particular, each time it encounters a call to a semantic #DF or #PROC-DF or an instance of a syntax #DF on the parse-tree for the program, it updates the result data base. A separate data base exists for each metaprogram archive/test suite archive pair. Each may be thought of as a matrix : each row represents the results of a particular test, and each column corresponds to the name of a syntactic #DF or semantic #DF (or #PROC-DF) in the SEMANOL specification. Each cell in the matrix records the number of times a given #DF occurred during a particular test.

Inquiring About Test Results in the Result Data Base

The reporting aspect of the Test Effectiveness Measurement Facility is provided by the ec inquire. Any submatrix of the result data base may be selected for display. This is done by providing lists of those tests (rows) and #DFs (columns) of interest. These lists are called Test Interest Groups (TIGs) and Specification Interest Groups (SIGs), respectively. For each test in the TIG, a list of all #DFs in the SIG is displayed, together with the number of times each such #DF was exercised. In addition, a cumulative list, including results for all tests in the TIG, is provided, together with a coverage percentage. The coverage percentage is derived from the following formula:

$$\text{coverage \%} = \frac{\text{\# elements in SIG with non-zero bit frequency}}{\text{\# elements in SIG}} \times 100$$

SUMMARY

This project was successful in developing a sound methodology for the systematic testing of compilers. The organization that was chosen for the JOVIAL(J73) Validator can be used in a Validator for any other programming language as well; in addition, this organization was utilized by the Test Effectiveness Measurement Facility. Likewise, the structure of the SEMANOL specification of JOVIAL(J73) developed in this project was heavily influenced by the needs of the Measurement Facility. Finally, the Measurement Facility that was developed can be used with any SEMANOL specification and any test set (of the required format).

This project also succeeded in producing an extensive Validator (i.e., 50,000 lines plus capacity tests) for JOVIAL(J73) compilers. This Validator tests many portions of JOVIAL(J73) very well and yet, despite its size, the Validator does not include adequate testing for other areas of the language; the 100,000 line Validator needed for truly thorough compiler testing was simply beyond the scope of this contract. In addition, the Validator itself has not been debugged. There are several reasons for this, which should be mentioned, for they are common to validator development efforts. First, TRW underestimated considerably the effort required to construct comprehensive tests of compilers for a language as complex as JOVIAL(J73); we originally would have expected a 50,000 line Validator to be quite complete. On the other hand, in our zeal to achieve "perfection," we sometimes found it difficult to maintain our perspective and engaged in "overkill," testing the same feature in many different, but largely redundant ways. This restricted emphasis came at the expense of coverage breadth. Finally, the Validator could not be suitably debugged because no translator for JOVIAL(J73) was available. Although a compiler was supposed to be provided for this project, the lack of a compiler is not an unusual occurrence, since the validator for a language is often developed prior to, or concurrently with, the first compilers for that language. The SEMANOL Interpreter system can be useful here, for it performs the same functions as a compiler, but it should be emphasized that confidence in the quality of a validator cannot be established without the availability of a proven processor.

One additional problem arose in trying to coordinate the Validator, the SEMANOL specification, and the Measurement Facility. The SEMANOL specification and the Measurement Facility were developed on the Multics computer at RADC. Originally, it had been planned to debug the test cases using the SEMANOL specification; then they too would have been developed on Multics. Instead, because of insufficient Multics resources, it was decided to debug the test cases using the JOCIT compiler. This compiler is hosted on an IBM 370; the only such machine available at TRW is a batch computer, not suitable for software development. Therefore, the test cases were developed on still another computer (actually two others, a Cyber system and a VAX 11-780). The logistics of using so many different computers sometimes became overwhelming, and emphasizes the desirability of doing software development and testing on the same computer, one that is a "friendly" host. (Some of the problems of the JOCIT compiler have been attributed to a similar situation).

There are several different future directions for this work. The Validator should be debugged, and it can also be enhanced to test lightly covered portions of the JOVIAL(J73) language more carefully. The SEMANOL specification of JOVIAL(J73) can undergo more thorough testing as well. Both can be upgraded to reflect MIL-STD-1589B, the current language standard for JOVIAL(J73). The Test Effectiveness Measurement Facility can be used to evaluate the Validator's quality with respect to the SEMANOL specification of JOVIAL(J73); this, however, would require enormous machine resources unless the speed of the SEMANOL Interpreter were greatly improved. The results of such an evaluation could then be used to enhance the Validator further. Finally, the Measurement Facility can be refined further to use a finer definition of structural element.

APPENDIX A
SEMANOL SPECIFICATION OF JOVIAL(J73)
DOCUMENTATION

This Appendix provides a high-level description of a specification of the JOVIAL(J73) programming language using SEMANOL(76) augmented by the TRY and FAIL-WITH-VALUE operators described earlier in this document and in the Revised SEMANOL(76) Reference Manual. Appendix B contains the text of this Specification; because SEMANOL is a very high level programming language, the Specification serves as its own detailed documentation.

Throughout Appendix A, SEMANOL code will be presented in a slightly modified notation for greater human readability.

- Keywords appear in upper case, without their distinguishing initial sharp-character # .
- Names of semantic definitions (DFs), parameters, global variables and local iteration dummy variables appear in lower case.
- Brackets ($ ... $) used for DF references in suffix notation, as in ($ arguments $) df-name, appear as [...] .

General Structure

The Specification requires that GIVEN-PROGRAM contain exactly those modules which comprise a single J73 program, since the SEMANOL Interpreter does not generate object code and cannot preserve parse trees between executions. Each module must be followed by a backslash-character \., and cannot contain any backslashes; any text following the last backslash in GIVEN-PROGRAM is available for substitution by !COPY directives, and is otherwise ignored.

The top-level structure of the J73 Specification is defined by its Control-Commands section.

```
CONTROL-COMMANDS:
    COMPUTE! compilation
    COMPUTE! execution
    COMPUTE! STOP .
```

The distinction between compilation semantics and execution semantics is mandated by the J73 concepts of separate compilation, values known at compile-time, and compile-time versus run-time error detection.

Compilation Semantics

The DF compilation and its subordinate DFs specify the compile-time processing for each module of a J73 program. This processing includes lexical analysis, parsing, and enforcement of constraints on individual modules.

```
PROC-DF compilation
BEGIN
     COMPUTE! init-program-compilation
     WHILE some-module-is-uncompiled DO
          COMPUTE! TRY compile-module ( text-of-next-module )
     RETURN-WITH-VALUE! NIL
END .

PROC-DF compile-module (text)
BEGIN
     COMPUTE! init-module-compilation
     ASSIGN-VALUE! module-tree =
          CONTEXT-FREE-PARSE-TREE
               ( lexically-transformed (text), <module> )
     IF [ module-tree ] is-uniquely-parsed THEN
          IF [ module-tree ] satisfies-module-constraints THEN
               COMPUTE! accept-module ( module-tree )
     RETURN-WITH-VALUE! NIL
END.
```

- init-program-compilation constructs values for the Specification global variables used during compilation, including "immutable" constants, implementation-defining constants, program-property constants, and initial values for compilation variables.
- some-module-is-uncompiled returns TRUE until all modules have been extracted from GIVEN-PROGRAM.
- text-of-next-module extracts the module from GIVEN-PROGRAM.

- init-module-compilation constructs global-variable initial values that must be renewed at the beginning of each module's separate compilation.

- lexically-transformed transforms the extracted module text into a lexically-equivalent canonical form, described in Lexical Transformation below.

- CONTEXT-FREE-PARSE-TREE (a SEMANOL operator) constructs a parse tree for the transformed module text according to the syntactic rules in the Context-Free-Syntax section of the Specification, described in Context-Free Parse below; is-uniquely-parsed confirms that the text was parseable as a J73 module.

- satisfies-module-constraints confirms that the parsed text conforms to all compile-time constraints on single modules, described in Constraint Enforcement below.

- accept-module adds the parsed and validated module to the global sequence variable seq-of-modules.

Each invocation of compile-module corresponds to the extraction, compilation and possible acceptance of a single module. To reject a module at any stage of compilation, the Specification invokes the DF compile-error, which prints an error message and evaluates a FAIL-WITH-VALUE expression to suppress the remaining compilation activities for that module. Since each invocation of compile-module is guarded in a TRY expression, the rejection of one module does not block the compilation of any further modules in the program.

Lexical Transformation

The DF lexically-transformed and its subordinates implement lexical analysis of a J73 program by transforming the text of a module into equivalent text in which all lexically-equivalent symbols have an identical canonical form. The particular transformations performed are listed below.

- A gap (blanks and/or newlines) between symbols is condensed to a single blank or newline character.

- A comment is replaced by a blank.

- A lower case letter is replaced by the corresponding upper case letter, except when it occurs in a character-literal or (prior to expansions of define-calls) a define-string or actual-define-parameter.

- A !COPY directive is replaced by text following the last backslash in GIVEN-PROGRAM; the exact replacement text is determined by the DF impl-copy-text reflecting implementation dependency.

- !SKIP, !BEGIN and !END directives cause suppression of text prior to parse and constraint enforcement, and are removed.

- Listing directives are replaced by a blank. (The Specification does not produce a paged source listing.)

- Define-calls are expanded; define-declarations are collected, applied, and replaced by blanks.

The use of scoped define-names for textual substitution in J73 assumes that the lexer and parser are operating as coroutines. The SEMANOL Parser, however, is a fully-encapsulated operator which is applied to a complete string. An exact model of J73 defines in SEMANOL would thus require reparsing a module after each define-call expansion therein, which would be prohibitively expensive. The current implementation correctly models define-semantics for all cases where the proper declaration of a define-call name is the textually-nearest declaration of that name preceding the define-call. Compool-importation of define-declarations is simulated by collecting all define-declarations into the global variable seq-of-define-decls, which is not reset between modules; redeclaration of a name deletes any existing declaration for it from this sequence.

Illegal recursion of define-calls is detected by maintaining in the global variable seq-of-active-define-names a list of those defines with expanded text currently being scanned. Each expanded define-string ends with a marker end-symbol (a backslash); when this mark is scanned, the last element in seq-of-active-define-names is deleted. If the define-name in a new call is already listed, an error is detected.

The overall structure of this lexical transformation is shown in the DF lexically-transformed below.

```
PROC-DF lexically-transformed (text)
BEGIN
     ASSIGN-VALUE! module-text = text
     ASSIGN-VALUE! scan-index = 1
     WHILE scan-index <= LENGTH(module-text) DO
          COMPUTE! transform-text-for-symbol
               ( symboll-in(unscanned-text))
     RETURN-WITH-VALUE! module-text
END.
```

The text of a single module extracted from GIVEN-PROGRAM is placed in the global variable module-text, for a single left-to-right scan. The DF symboll-in finds the next symbol being scanned. The DF transform-text-for-symbol then determines whether this next symbol is in canonical form; is so, the scan continues with the text following this symbol; if not, the canonical form of the symbol is substituted, and the scan is repeated on the substituted text.

Context-Free Parse

The parsing phase and later phases of module compilation semantics expect that the text submitted to the Parser has been passed through lexically-transformed.

The SEMANOL syntax for a J73 module generally follows the structure of the syntax in MIL-STD 1589A; in some areas, however, the Standard's grammar is overloaded in order to simplify structure, represent contextual constraints, or increase readability. The SEMANOL grammar must diverge in such areas, in order to provide unambiguous parsing of all legal programs. For example, the SEMANOL grammar contains only one derivation for each kind of arithmetic expression, distinguished by the arithmetic operator; the type class of the result is explicit in the Standard's grammar, but is generally dependent on the type class of the operands, which may be item names or function calls whose type classes are contextually determined.

Syntactic analysis of a J73 program must resolve the classical problem of the ambiguity of the optional ELSE within an IF statement:

```
IF a THEN
IF b THEN s
ELSE t
```

Is t performed when (NOT a) or when (a AND NOT b) ?
The Standard uses a prose description to match the ELSE with the innermost unmatched IF; in this case, (a AND NOT b). The SEMANOL grammar disambiguates such statements formally, by dividing all statements into two classes, open and closed. Open statements end in a THEN clause with no matching ELSE; all other statements are closed, and the conditional-statement in a closed IF statement is forced to be a closed statement.

Another possible ambiguity is introduced by a program-body or subroutine-body other than a single statement, which is parsed

... [<declaration> ...] <statement> .....

Between the constructs which are definitely declarations and those which are definitely statements, there may be one or more constructs that are null "statements" (textually identical to null "declarations"), or compound structures containing only nulls and other such compound structures, without labels that would identify them as definite statements. The Standard's grammar and text do not disambiguate the syntax of programs containing such body fragments, since such null constructs have no semantic significance and thus introduce no semantic ambiguity. The SEMANOL Parser has no foreknowledge of such convergence of meanings, and cannot tolerate such ambiguities; therefore, the Specification syntax for J73 body constructs has been "biased" to begin the sequence of statements only with a construct that is clearly a statement rather than a declaration. The opposite bias would not work, since each body is required to contain at least one nonnull statement, but is not required to contain any nonnull declarations. The body is then parsed

... [<declaration> ...] <definite-statement> [<statement> ...] ...

where a label or a non-null non-compound component statement is sufficient to identify a definite-statement.

The Specification syntax contains some special syntactic classes that always produce the NIL string of terminals; these classes are used to insert marker nodes into parse trees for simplified specification of flow-of-control

semantics. Such classes include end-body, end-conditional-statement, end-if-statement, end-while-statement, end-for-statement, end-case, and end-case-statement. Their significance will be described in Program Flow of Control.

Constraint Enforcement

The constraints on J73 programs can be divided into five major groups, corresponding to the phases of compilation and execution in which they can be detected. Each section of the Specification incorporates the enforcement of certain constraints appropriate to that section. Because effort has been concentrated on executing correct programs rather than on rejecting incorrect programs, not all J73 constraints are enforced by the current Specification; however, the predicates necessary to specify full enforcement are provided. Full enforcement of module constraints alone for a complexly-scoped language such as J73 is expected to use more than 25% of the total processing time required by a representative test set.

Constraints such as the prohibition on recursive define-calls are enforceable during lexical analysis. In the Specification, violations of such constraints are detected in lexically-transformed, which then invokes compile-error to print a diagnostic message and FAIL-WITH-VALUE out of the compilation of the current module.

Constraints like the disambiguations of IF-ELSE and of null declaration-statements, described in Context-Free Parse above, are enforceable as context-free syntax rules. In the Specification, such constraints are represented in the CONTEXT-FREE-SYNTAX section used by the CONTEXT-FREE-PARSE-TREE operator to parse a module. Violation of these constraints causes the parsing operator to return the value UNDEFINED; is-uniquely-parsed then invokes compile-error to print a 'module unparseable' diagnostic message and FAIL-WITH-VALUE out of further compilation of the module.

Other constraints enforceable within a single module, such as the requirement that a name can only be used within the scope of a unique declaration of that name, are also part of compilation semantics. In the Specification, such constraints are enforced by the DF satisfies-module-constraints, which consists of the evaluation of a series of module properties, as seen below.

```
DF satisfies-module-constraints (mt) "mt EQ module-tree"
=> all-names-are-declared (mt) &
   all-formulas-have-unique-types (mt) &
   all-ct-formulas-have-ct-values (mt) &
   all-ct-values-are-valid (mt) .
```

The evaluation of each property includes invocation of compile-error whenever a constraint violation is detected. The list of properties currently enforced by the Specification does not cover all constraints required by J73 at the module level, but does ensure that the program has a sound basic structure; enforcement of the remaining constraints can be accomplished by uncomplicated extension of the current Specification. (The type equivalence and type compatibility relationships are defined in the current Specification and discussed below in Types, although they are not yet used in constraint enforcement.)

The enforcement of module constraints includes the critical "first references" for most syntactic-component DFs (DFs identified in the DECLARE-SYNTACTIC-COMPONENT section of the Specification as constant properties of nodes in program parse trees). The first reference to such a DF for a given node-argument evaluates the DF-body with that argument, and then tags the node with the result; all further references to that DF with the same argument will pick up this saved value, without re-evaluating the DF-body. The syntactic component ct-value is used to achieve the J73 concept of a "value known at compile-time", as described in Formula Evaluation below. Various other syntactic components optimize the derivation of constant properties of program parts, as discussed in Scope of Names and in Types below.

Constraints such as the need for exactly one main-program-module in a program cannot be enforced during separate compilation of each module; they must be enforced during linkage of a collection of modules into a program. In the Specification, these constraints are enforced by the DF program-satisfies-constraints, which consists of the evaluation of a series of program properties, as seen below.

```
DF program-satisfies-constraints
=>  TRUE IFF
        all-modules-were-successfully-compiled &
        exactly-one-main-program-module-was-compiled &
        no-external-name-was-multiply-defined .
```

The evaluation of each property includes invocation of link-error whenever a constraint violation is detected, to produce a diagnostic message and suppress execution of the erroneous program.

Constraints such as range restrictions on non-compile-time values can only be enforced "dynamically", during program execution. In the Specification, these constraints are enforced within the definitions of evaluation, storage and control semantics. When a violation is detected, fatal-error is invoked to print a diagnostic message and terminate execution.

Scope of Names

J73 provides a complex mechanism for associating names with textual entities in a program, including the concepts of system-defined names (implementation parameters and machine-specific subroutines), external names for objects defined in one module and used in another, importation of declarations of names from compools, and nesting of name scopes. The Specification implements these concepts in terms of the DFs declaring-occurrence-of, scope-of, and scoped-name-of.

Nodes of the syntactic class <name> , of any of the classes which produce <name> directly (e.g. <program-name> , <type-name>), are considered name-nodes; they satisfy the predicate [n]is-a-name. The declaring-occurrence-of any name-node is a name-node yielding the same textual name, occurring in a

declaration which determines the nature of the named entity and the scope of the name. Every declaring-occurrence name-node is a member of one of the distinguished syntactic classes:

item-name
table-name
block-name
constant-item-name
constant-table-name
type-name
statement-name
label-name
subroutine-name
compool-name
program-name
control-letter
system-scope (for system-defined names).

The scope-of a name-node is a node identifying a scope, and is a member of one of the syntactic classes:

system-scope (for system-defined names)
module (for the compool scope in compilation of a module)
module-scope (for the module as a scope)
program-body
subroutine-definition
subroutine-declaration
(open/closed) for-statement (for loop-control-letters)

Finally, the scoped-name-of a name-node constructs a SEMANOL string value unique to the textual object named by the node. In general this string consists of the scoped-name-of (the scope-object enclosing the declaration of the named object), concatenated with a colon and the textual name of the object.

- scoped-name-of (the system-scope) is the NIL string.
- scoped-name-of (the n^{th} module in the program) is ':MOD,n', which is unique since MOD is a reserved word in J73.
- scoped-name-of (the program-body) is ':PROGRAM'.
- scoped-name-of (any external name) is ':DEF:text-of-name', so that all DEFed and REFed declarations of a name refer to the same object.

During execution, the Specification will need to be able to map back from scoped-names to their defining occurrences; therefore, during program linkage, the defining-occurrence node for each scoped-name is stored in the SEMANOL LV-space as the LATEST-VALUE of the scoped-name string. (Most declaring occurrences, except REFed declarations and formal-parameters of subroutine declarations, are defining occurrences.)

The current Specification accurately models all aspects of J73 scopes, except the implicit importation of certain names due to explicit importation of other names (e.g. importation of a type-name when importing an item-name declared with that named type). No inherent difficulty prevents the extension of the Specification to model these omitted aspects of importation.

Types

J73 is a "strongly typed" programming language, in that every name or expression in a J73 program has a definite nature (a type class and set of attributes), which restricts the operations and contexts in which it can appear, and which can only be overruled by explicit redefinition of this nature to another specific nature (explicit conversion). This nature is determined by rules which combine component types according to the form of the expression.

The concept of type is so pervasive in J73 that the Specification DF type-of constructs SEMANOL sequence-value representations of the type of each name or expression or item-description, where the concept has been extended to include procedure types (including parameter lists and subroutine attributes), and type-representations themselves as named types. The various possible forms of type representations are listed below, with the corresponding attribute names. In the Specification, the DFs which decompose such representations to retrieve attribute values are named by (attr-name)-of-type(type-rep).

```
Item types:

class: 'S' or 'U' (integer)
size: INTEGER
rounding: 'R' or 'T' or NIL(default)

class: 'F' (floating-point)
precision: INTEGER
rounding: 'R' or 'T' or NIL(default)

class: 'A' (fixed-point)
scale: INTEGER
fraction: INTEGER
rounding: 'R' or 'T' or NIL(default)

class: 'B' (bit)
size: INTEGER

class: 'C' (character)
size: INTEGER

class: 'P' (pointer)
basis-type:  type-name node or UNDEFINED

class: 'STATUS'
size: INTEGER
has-default-reps: BOOLEAN (TRUE if no specified-sublist in status-list)
seq-of-status-values: SEQUENCE-OF text-of(status-constants)
seq-of-status-reps: SEQUENCE-OF INTEGER


Table types:

class: 'TABLE'
is-ordered: BOOLEAN (TRUE if order-directive present)
seq-of-table-components: SEQUENCE-OF
  \component-node, packing \ for an ordinary entry, or
  \component-node, starting-bit, starting-word \ for a specified entry
      ( component-node is
            <table-item-name>  where present,
            <item-type-description> in bodiless tables;
        packing is 'D' or 'M' or 'N';
        starting-bit and starting-word are INTEGER )
table-structure: 'PARALLEL' or 'T' or 'Tn' or NIL (default serial)
      ( n in 'Tn' is INTEGER )
seq-of-dimensions: SEQUENCE-OF <dimension>
words-per-entry: NIL for an ordinary table, or
      'V' or 'W' or 'Wn' for a specified table
```

Block types:

class: 'BLOCK'
is-ordered: BOOLEAN (TRUE if order-directive present)
seq-of-block-components: SEQUENCE-OF declaring-occurrence-of(components)
    (excludes components of components)
seq-of-block-overlays: SEQUENCE-OF <overlay-declaration>

Named types:

class: 'TYPE'
named-type: SEQUENCE (representation of a type)

Subroutine types:

class: 'PROC'
return-type: SEQUENCE
    (representation of a type for functions, or NILSEQ for procedures)
seq-of-input-parms: SEQUENCE-OF declaring-occurrence-of(input-parameters)
seq-of-output-parms: SEQUENCE-OF declaring-occurrence-of(output-parameters)
multiplicity: 'REC' or 'RENT' or NIL
is-pure: BOOLEAN (TRUE if reducible-directive is present)

J73 defines a few special kinds of expressions whose types are determined by the contexts in which they occur.

- A real-literal takes the fixed or floating type determined by its context, if there is one; otherwise it takes the default floating type.
- A status-constant belonging to more than one visible status type takes the type determined by its context, if there is one; otherwise it is illegally ambiguous.
- A pointer-literal NULL in an assignment context must take the named basis-type attribute of this context, if there is one (since an untyped pointer must be explicitly converted before being assigned to a typed pointer); otherwise it is an untyped pointer.

The Specification handles all of these aspects using the DF context-type, which for a given expression node contains any definite type inherited from the context of the expression, and otherwise is UNDEFINED.

- context-type(an operand of a binary
  relational or arithmetic operation) =
  type-of(the other operand)
  if the other operand has a type determined independent of its context.

- context-type(an assignment or preset formula) =
  type-of(the object being given a value).

- context-type(an actual parameter) =
  type-of(the corresponding formal parameter).

- context-type(a loop initial value) =
  type-of(the loop control-item)
  if the loop is controlled by a named variable
  rather than a control-letter.

- context-type(a value being explicitly converted) =
  the result type of the conversion.

- context-type(a table subscript index) =
  the type of the corresponding dimension.

- context-type(a dimension bound) =
  type-of(the other bound of the dimension)
  if the other bound has a type determined
  independent of its context.

- context-type(a case-index) =
  type-of(the case-selector of the case-statement)
  if the case-selector has a type determined
  independent of its context.

Representation of Values

Unlike many other HOLs, J73 incorporates some definite ideas about how data values are stored and represented. Representations are constrained by the assumption that storage of data values is organized in fixed-length bit strings called words, which are identified by ordered addresses that can be mapped to a subset of the integers. One effect of this assumption is that every value is represented by a bit string; thus, each item type has at least one attribute specifying the number of bits required to represent values in that type (or for character data, the number of k-bit bytes). Explicit conversions are defined between bit types and nonbit types, which rely on the underlying bit string representations of all values. Data of one type can be stored in a variable of that type, and this storage can then be addressed under an "alias" as a variable of another type. Although programs using such effects are in general "unsafe", a J73 program may use these effects in safe ways which should yield the same result for all J73 implementations.

The SEMANOL machine does not itself conform to J73's architectural assumptions. The elements of its storage model have no intrinsic order; its address space is its text string domain rather than only its integer strings; and its storage elements can contain values from its sequence and parse-tree node domains, as well as strings of any length. Its machine operations are generally defined in terms of operands from these varied domains, rather than assuming a common underlying bit string representation. Thus, in order to accurately interpret all J73 programs using the SEMANOL machine, some intermediate levels of machine-architecture must be modelled. These levels should include an "implementation"-dependent mapping between the SEMANOL representations of J73 bit string values and the SEMANOL representations of all other J73 data values; and an "implementation"-dependent method of allocating storage to data variables, where the storage is modelled by using the SEMANOL representations of J73 bit strings (of a single length for a given "implementation") as the LATEST-VALUEs for SEMANOL representations of J73 pointer values.

The J73 features whose semantics rely on such architectural assumptions are primarily those which provide ways for a J73 program to retrieve or modify a value assuming a type for the value which differs from the type under which the

value was originally stored. These features include:

- overlay-declarations -- directly associate multiple names with the same area of storage, or control relative positioning to justify later pointer manipulations.
- specified tables -- may overlap table-items within an entry, or variable-length entries within a table, or may control relative positioning within a table or entry.
- REP pseudovariables -- store any bitstring as the bit pattern representation of the value of the variable, even if the bitstring cannot be interpreted as a valid value of the type of the variable.
- NEXT of a pointer -- generate a pointer of the same type as the argument by integer-address displacement from the argument pointer; the result may be a typed pointer that could be dereferenced to designate an object of the basis type, even if no such object is located at the given displacement.
- conversions from nonpointer types to pointer types -- may be dereferenced to designate an object not of the assumed type.
- coercion of a pointer to a typed pointer -- may be dereferenced to designate an object not of the assumed type.

Specification of the full semantics of such features represents an order of magnitude increase in the complexity of the Specification, and was therefore beyond the level of effort available in the current project. In order that the Specification might be of use for the Measurement Facility, it was considered preferable to develop a Specification which could execute programs using the "safe" features of J73, rather than to produce only a design for a complete Specification. The "unsafe" features are included in the Specification grammar, and to some extent in the compilation of each module, so that coverage of these features by test sets can be partially assessed.

The result of this decision is that the current Specification contains a limited model of storage and data representation in which the choices for data representation are tailored to the SEMANOL machine as both host and target machine for a "canonical" implementation of J73. The data representations for values of the various type classes are described below.

Numeric values are represented by SEMANOL strings which look like the literals for the corresponding type classes. A subgroup of "canonical" representations is also defined for each of these type classes, and the Specification expects that representations returned by the DF value will have this canonical form.

- Integer values are represented by strings of the SEMANOL domain INTEGER; the canonical form has no leading zeros and no positive sign.

- Floating-point values are represented by strings with the form of a J73 real-literal; the canonical form has no decimal point, and its mantissa and exponent contain no positive sign and no leading or trailing zeros.

- Fixed-point values are represented by strings with the form of a J73 real-literal; like floating-point values, the canonical form contains no positive sign and no leading or trailing zeros, but unlike floating-point, the canonical form eliminates the exponent rather than the decimal point.

Character values are represented by SEMANOL strings, where the LENGTH of the representation equals the character-size of the value. This means that the representation of a character-literal excludes the delimiters and doubling of delimiter-like characters that are present in the literal-form.

Bit values are represented by strings of the SEMANOL domain BITSTRING containing the same number of bits as specified by the bit-size of the value. This means that the representation of a bit-literal expands the bead-string for any bead-size greater than 1, and that the representations of boolean-literals are the representations of the equivalent bit-literals.

Status values are represented by the representations of their corresponding integer "representational values", as defined by J73. This choice of representations (rather than the texts of status-constants, for example) is "safe", since J73 constraints ensure that the type of a status value is always known wherever the value is accessible; and this choice best reflects the Standard's definition of all status relationships in terms of the relationships between their representational values. Certain relationships appear to be defined by the Standard in terms of the textual order of status-constants in

their status-list, including bounds of a status dimension and NEXT functions for a status argument; however, constraints restrict these cases to status types with default representations, for which textual order is equivalent to the integer order of the representational values.

Pointer values are represented by SEMANOL strings corresponding to "addresses" in the SEMANOL machine's associative store (the LV-space). The string 'NULL' represents the value of the pointer-literal NULL, and is rejected as an inaccessible address by the value-storage and value-retrieval DFs put-value and get-value (described in Data Storage below). Otherwise, a pointer to a data object is represented by the "standard-name" string unique to that data object, whose construction is described in Data Storage below.

Table values are represented in storage as the collection of their table-item components. The value of a table formula, appearing in table assignments, is represented by the standard-name unique to the table or table-entry object designated by the formula. Table assignment is thus implemented by reference semantics, in that the evaluation of a table expression returns a pointer to the table, rather than an aggregate value containing the values of all components of the table. The Standard specifies value semantics for all assignments; this departure in the Specification avoids the complication of constructing representations of aggregate values, and is correct for all table assignments except those of the form

```
@ ( ptr'function (arguments) ) = table'expr ;
```

or

```
table-name ( index'function (arguments) ) = table'expr ;
```

where the evaluation of ptr'function or index'function alters the value stored at the address of the table'expr object.

Block values are not represented directly, since J73 never introduces block values independent of the block objects which have those values; thus block parameters are adequately represented by their standard-name addresses.

Execution Semantics

The DF execution and its subordinate DFs specify the non-compile-time processing for a J73 program as a whole. This processing includes linkage of a collection of modules into a program, loading the program for execution, and interpreting the program by "invoking" the program body as a routine from the "System".

```
PROC-DF execution
BEGIN
   IF TRY program-satisfies-constraints THEN
      COMPUTE! accept-program
   IF NOT execution-suppressed THEN
      COMPUTE! invoke-program
   RETURN-WITH-VALUE! NIL
END .
```

The DF program-satisfies-constraints enforces linkage constraints. If the program is accepted, then it is "linked" in accept-program through the LV-space: each defining-occurrence of a name is saved as the LATEST-VALUE of the scoped-name string unique to that definition; thus, during program execution, the DFs using a given scoped-name can "look up" the meaning of that name. The global variable execution-suppressed is FALSE if and only if all modules of the program were successfully compiled and the program as a whole satisfies all linkage constraints; otherwise interpretation of the program is blocked.

The DF invoke-program constructs values for Specification global variables used during execution, creates the system environment from which the program will be invoked, "loads" the J73 program by assigning initial values to all preset data objects, invokes the program-body as a routine from the system environment, and reports the completion (or termination with possible stop-value) of the program.

The detailed semantics of program execution are discussed below, in Formula Evaluation, Data Storage, and Program Flow of Control.

Formula Evaluation

The DF value and its subordinate DFs specify the semantics of formula evaluation. The result constructed by value for a given formula is a value-representation in the canonical form for the type class of that formula, as described in Representation of Values above.

```
DF value (expr)
=> ct-value(expr)  IF  [expr]is-ct-value ;
=> new-value(expr)  OTHERWISE .

DF ct-value (expr)
=> new-value(expr) .
```

The DF new-value constructs the current value of a formula. The predicate is-ct-value identifies J73 formulas which produce constant "values known at compile time"; ct-value is declared as a syntactic component, so the new-value of each such formula is constructed only once, is saved on the parse-tree node for that formula, and is used in all further references to the value of that formula.

An important general-purpose DF used in evaluations is value-conformed-to-type, which takes a value and an expected result type, verifies that the value is a member of the type, and converts the value to its canonical representation for that type. This representation may be an approximation of the value, due to loss of precision specified by the type; if the type does not contain even an approximation of the value, a fatal-error is detected. This DF is also used to verify the legality of values in contexts involving implicit conversion, such as assignments and presets.

Evaluation of integer, bit and character literals cannot use the DF value-conformed-to-type to check that the literal represents a valid value of the formula type, because the size-attribute of such a formula is determined by its value, rather than the value being constrained by a type. Instead, the value is confirmed to have a size no greater than that of the largest legal literal of its type class. Real-literals, pointer-literals, and status-constants, on the other hand, have types determined by their contexts, as discussed in Types; thus a real-literal can be filtered through

value-conformed-to-type safely.

The evaluation of each arithmetic operation is subdivided first according to the type class of the result (integer, fixed or floating), and then according to whether the implementation being modelled is the "canonical" implementation for the SEMANOL machine itself. Other implementation models are expected to differ in their representations of data values; such representations and the arithmetic on these representations must be defined when alternate implementation models are incorporated into the Specification. The result of each arithmetic operation is submitted to the DF value-conformed-to-type, along with the specified result type, to yield the canonical representation of a value in the result type.

The evaluation of the logical-continuation operations reflects the Standard's requirement to determine the result size based on the sizes of all N operands in the expression; thus, evaluation of an N-operand continuation is not equivalent to successive evaluation of binary logical operations. For example,

```
    1B'0'   EQV   1B'1'    EQV  1B'11'  has value  1B'10';  also
   1B'00'   EQV  1B'01'    EQV  1B'11'  has value  1B'10';  but
( 1B'0'   EQV   1B'1' )  EQV  1B'11'  has value  1B'00'.
```

As a special case, short-circuit evaluation applies only when all N operands are boolean values; if the above examples were ANDs instead of EQVs, short-circuit evaluation could apply only to the evaluation of the parenthesized subexpression in the last example.

The relational operations, like arithmetic, depend on the forms of value representations, and are therefore subdivided first by the result type class, and then by the implementation being modelled. Only the "canonical"-implementation operations are defined in the current Specification. In these operations, character comparisons are based on the order of single characters in the global variable impl-collating-sequence. Pointer comparisons are based on three assumptions:

- static data precedes automatic data;
- automatic data of a given invocation precedes automatic data of all later invocations;
- pointers to data allocated in the same invocation are ordered according to character-comparison of their standard-name representations.

The LOC intrinsic function is implemented as the standard-name identifying the argument object; and is thus limited only by the Specification's ability to construct such an identification for the object. The NEXT function is fully implemented for status arguments; however, because the NEXT function for pointer arguments is so heavily dependent on detailed modelling of an implementation's storage allocation methods, this function currently returns the NULL pointer value (to block dereferencing of a possibly-meaningless typed pointer). The Specification could be extended to return meaningful pointers where this function is being used "safely", as in stepping through the entries of a table; but this extension proved to be beyond the level of effort available for the current project.

The SHIFTL and SHIFTR functions, the ABS and SGN functions, and the FIRST and LAST functions are fully implemented. The LBOUND and UBOUND functions are fully implemented, including the non-compile-time aspects associated with *-dimension formal-parameter tables. The NWDSEN function is completely defined in terms of the BITSIZE of the table entry. BYTESIZE and WORDSIZE are also fully defined in terms of BITSIZE.

BITSIZE of an item or item type is fully defined, including the distinction between signed and unsigned integer representations of status types. BITSIZE of a table or table type distinguishes between tight entries, other specified entries, and other ordinary entries. Because a fully detailed storage model was classed as beyond the available level of effort, certain special constraints on alignment of table components for byte and word boundaries are not considered in constructing BITSIZE.

The BIT and BYTE functions are fully implemented as decompositions of the values of their primary bit-formula or character-formula arguments. Since the current Specification does not consider "filler bits" and other differences between logical and physical representations of data, the REP function is implemented using a bit-type-conversion.

The Specification fully implements explicit conversions between two types of the same type class, except for those pointer conversions that assume the colocation of non-equivalent objects (such conversions provide a form of "aliasing" dependent on a detailed storage and value-representation model). All explicit conversions between two numeric types are also fully implemented, since these associations of numeric values in different classes are essentially independent of particular representations. The remaining J73 explicit conversions make some assumption of either an underlying bit-string representation of all data values, or a correlation between pointer values and integers. Since these assumptions are not supported in the current Specification, the remaining item-type conversions have been defined to return values of the proper result type, regardless of the value being converted as follows:

- conversions to numeric types return zero;
- conversions to bit types return the bit-value FALSE padded to the proper length;
- conversions to character types return a string of blank characters of the proper length;
- conversions to pointer types return the pointer value NULL;
- conversions to status types return the value of FIRST for that status type.

Conversion of a bit value to a table value is UNDEFINED in the current Specification.

The remaining kinds of J73 formulas have all been parsed as <named-reference>, since their exact natures depend on the declaration of names appearing in the formulas. Such a formula may be:

- an implementation parameter, whose value is defined by a correspondingly-named DF in the Implementation Dependency section of the Specification;
- a call to a user-defined or machine-specific function, whose semantic significance is given by the DF invoke-function described in Program Flow of Control;
- an actual-parameter that is to be bound by reference (a block or table data object, or a label or subroutine name that must identify the dynamic environment of routine invocations as well as the textual object being named); or
- a loop-control-letter or named-variable, whose current value is retrieved by

  get-value(standard-name(named-reference))

  (get-value and standard-name are described in Data Storage).

Data Storage

As discussed in Representation of Values, the SEMANOL machine architecture does not conform to J73's assumptions about the organization of storage of data values. The current Specification implements a relatively abstract storage model appropriate to the SEMANOL machine organization and to the level of effort available in the current project. This model is adequate for most "safe" programs (programs that do not depend on a particular implementation).

A defining-occurrence of a data-name identifies a distinct "textual data object". If a textual object has STATIC allocation, exactly one instance of this object will exist throughout execution of the J73 program. If a textual object has automatic allocation, then each time the subroutine enclosing the defining-occurrence is invoked, a new instance of this object will be allocated storage; this new instance exists only until the invocation is completed or terminated.

Each allocated instance of each textual data object in a J73 program is designated by a "standard name" unique to that instance of that object, constructed by the DF standard-name at each dynamically-interpreted reference to that object. Each standard name is built up from the scoped-name unique to a given independent textual data object (an object that is not a component of any other data object). A "dynamic prefix", consisting of a canonical integer followed by a hyphen, identifies the particular routine invocation for which this independent data object was allocated ('0-' is the prefix for static data objects). A component object is identified by adding a "component selector", consisting of a dot-qualification suffix added to the standard-name of the aggregate object containing the component. The n^{th} table-item in a table entry, or the n^{th} component of a block, uses the suffix '.n-1'. Each entry of a table object uses a normalized form of its subscript, replacing each subscript-index by a canonical integer representing the offset of this index value from the lower-bound of the dimension. As an example, consider the code fragment below.

```
START PROGRAM PP; BEGIN ...
   PROC QQ; BEGIN
      BLOCK BB; BEGIN
         TABLE TT ( 0:2, 1:3, LAST(STAT'TYPE) ) ; BEGIN
            ITEM WW S ;
            ITEM XX S ; ...END ... END
      XX( 0, 1, FIRST(STAT'TYPE) ) = ... ; END ... END TERM
```

During a single invocation of QQ that is the fifth routine invocation in program execution, the standard name of the variable being assigned a value is '5-:PROGRAM:QQ:BB.0.(0,0,0).1'.

'5-' denotes the invocation of QQ as the fifth routine invocation;

':PROGRAM:QQ:BB' is the scoped name denoting the independent block BB declared in QQ;

'.0' selects the first component of BB (which is the table TT);

'.(0,0,0)' selects the first entry of TT ; and

'.1' selects the second table-item in this entry.

Component and entry selectors are normalized because table types are equivalent when they have the same structure, even if component names are not identical, and when they have the same number of index values in each dimension, even if the dimension bounds are not the same and even if one table has a status dimension where the other has an integer dimension. Thus the normalization of component selectors simplifies table assignment and reference binding of table parameters. Most compilers perform analogous processing for component objects, computing an offset from the aggregate address to compute a component address; the difference is that the compiler then adds the offset to the integer aggregate address, while the Specification concatenates the offset onto the SEMANOL string aggregate address.

The standard name of an item data object identifies a single storage element in the SEMANOL machine LV-space; the LATEST-VALUE of this standard name yields the representation of the current value of the item. The item may be an independent object, or a table-item component of a table, or an item component of a block. The current value of a block or table object is the collection of the current values of its component objects.

The creation of static data objects at the start of program execution is implemented by the DF create-preset-data. Uninitialized objects need not be explicitly created, since the Specification detects a "dangling pointer" (an attempt to access a nonexistent data object) by checking the dynamic prefix in the object's standard name against a list of current invocations, rather than by detecting some distinguished value in unallocated storage.

The DF get-value implements the semantics of retrieving the current value of a data object. It takes a standard name as the designation of the object, verifies that this object currently exists, and returns the current value of this object filtered through value-conformed-to-type to protect against the use of aliases of different types for the same storage. Existing objects which have not yet been given a definite value have the SEMANOL value UNDEFINED; attempts to retrieve the value of such an object will invoke fatal-error.

The DF put-value implements the semantics of giving a value to a data object. It takes a standard name designating the object and a representation of a value, verifies that this object currently exists, and stores the value representation as the current value of the object. Values are filtered through value-conformed-to-type to ensure that the type of a stored value agrees with the type of the variable in which it was stored; this provides for the implicit type conversions specified for J73 assignment semantics (including presets and item parameters).

Assignment to a table object is implemented by component-wise assignments to its table-items. Assignment to a BIT or BYTE pseudovariable is implemented by retrieving the current value of the entire variable to be modified, reconstructing this value around the modified substring value, and assigning the result to the entire variable. Assignment to a REP pseudovariable is implemented by substitution of SEMANOL underlying string representations, rather than by substitution of J73's assumed underlying bit representations.

The LATEST-VALUE of the standard name of a formal parameter bound by reference contains the standard name of the corresponding actual parameter for the particular routine invocation (containing this instance of the formal parameter) in question. get-value and put-value are each responsible for implementing reference binding for these parameters; when they are given the standard name designating a reference parameter, they look up the standard name of the corresponding actual parameter, tracing back through a possible chain of parameter associations to find the standard name designating the actual object to be referenced; they then perform the value retrieval or assignment on that actual object.

Program Flow of Control

The Specification describes the semantics of flow of control within a J73 program in terms of certain basic concepts: the routine, the executable step, and the locus of control. A routine is either a user-defined procedure or function, or the program body (which can be considered as a subroutine invoked by the "system"). The body of each routine is interpreted as a sequence of executable steps, which are parse-tree nodes corresponding to statements or to certain control-points within statements. Execution of the body of a routine consists of the successive application of the effect of each executable step in this sequence, starting with the first step in the body, where the effect of a step may change the value of one or more program variables, cause the invocation of another routine, or redirect the flow of control within the current invocation so that the next step executed is not the natural textual successor of the current step.

A locus of control has two parts: a sequence representing an environment, which identifies the invocations whose allocated data objects are directly visible by name at this point in execution, and an executable step, which must be an element in the body of the "current" invocation (the last invocation identified in the environment). Each new invocation is identified by a unique canonical integer, which represents the invocation in environments, and which serves as the distinguishing dynamic prefix in the standard names for automatic data allocated in the invocation. Each environment is constructed with invocation numbers parallel to the concatenation of scopes in scoped-names, simplifying the determination of dynamic prefixes for data references.

The specification of flow-of-control semantics has been divided into two major levels: the routine-invocation level, involving changes to cur-env (the environment part of the current locus of control), and the executable-step level, involving changes to cur-step (the executable-step part of the current locus).

Routine-level semantics includes parameter bindings and either normal completion (by RETURN) or abnormal termination (by STOP, ABORT or nonlocal GOTO) of invocations, as well as the actual invocation of the program-body or a subroutine. It is specified by three major DFs:

- invoke-program invokes the program-body as a routine called from the "system" environment, with no actual parameters and with the system locus of control as the implied destination for any ABORT in the program-body;

- invoke-procedure invokes a user-defined procedure due to the execution of a procedure-call statement, after first selecting and evaluating the actual parameters for this call, and supplying either the ABORT-destination specified in the procedure-call statement or the destination inherited from the invocation containing this statement;

- invoke-function similarly invokes a user-defined function during the evaluation of a function-call expression, after selecting and evaluating the actual parameters, and supplying the ABORT-destination inherited from the invocation calling the function; after normal completion of the function, invoke-function retrieves the result value of the call and returns it to the evaluation-semantics DF value-of-function-call.

The selection and evaluation of an actual parameter yields a sequence value in one of the following forms.

- An item input actual parameter (bound by value) consists of
  (1) the NIL string in lieu of the object's standard-name designation, which has no significance for value parameters, and
  (2) the representation of the current value of the object.

- An item output actual parameter (bound by value-result) consists of
  (1) the standard name designating the actual-parameter object, to allow assignment of result value on completion of the new invocation, and
  (2) the representation of the current value of the object.

- A block or table actual parameter (bound by reference) consists of
  (1) the standard name designating the actual-parameter object, and
  (2) the NIL string in lieu of the object's current value, which has no significance for reference parameters.

- A label actual parameter consists of
  (1) the current environment, which would be the destination environment for a GOTO to the formal-parameter label name, and
  (2) the defining-occurrence of the actual label.
- A subroutine actual parameter consists of
  (1) the nonlocal environment for an invocation of the formal-parameter subroutine name, and
  (2) the first executable-step node in the body of the actual subroutine.

The representations for subroutine and label parameters reflect the fact that in J73 such parameters have "deep binding"; they carry a particular environment to be recovered when the formal parameter is used, rather than simply identifying a textual entity whose most recently created instance is to be used. The ABORT-destination has a similar deep binding, and is constructed and used as if it were a special label parameter.

The general semantics of routine invocation is specified by the DF invoke-routine, which is called by each of the three invocation DFs named above.

```
PROC-DF invoke-routine (calling-locus, called-locus,
                        arguments, abort-dest)
BEGIN
   COMPUTE! set-current-locus-to( [called-locus]with-new-invocation )
   COMPUTE! set-cur-abort-dest( abort-dest )
   COMPUTE! set-formal-parameters-from( arguments,
      seq-of-formal-parameters-in( called-routine( called-locus )))
   COMPUTE! execute-invoked-body( cur-locus )
   COMPUTE! set-result-arguments-from( arguments,
      seq-of-formal-parameters-in( called-routine( called-locus )))
   IF  [ called-routine(called-locus) ]is-a-function  THEN
      ASSIGN-VALUE! function-return-value =
         LATEST-VALUE( function-return-value-name )
   COMPUTE! deallocate-cur-invocation( void-locus )
   COMPUTE! set-current-locus-to( calling-locus )
   RETURN-WITH-VALUE! NIL
END .
```

The DF with-new-invocation uses the nonlocal environment supplied in called-locus, and adds a unique integer for the new invocation; this cannot be done until after the selection and evaluation of actual parameters to the invocation, which could include function calls invoking other routines. Similarly, the specified abort-destination for the new invocation does not apply to function calls in the actual parameters.

The DFs whose invocations follow execute-invoked-body within invoke-routine represent the effect of normal completion of the routine body upon execution of an explicit or implicit RETURN statement; when abnormal termination of the routine occurs, execute-invoked-body for the routine is set aside by the effect of the SEMANOL operator FAIL-WITH-VALUE (as described below), and these normal-completion effects are not performed. The DF deallocate-cur-invocation removes the current invocation's number from the list of invocations with existing automatic data, as part of the mechanism for detecting dangling pointers.

```
PROC-DF execute-invoked-body ( called-locus )
BEGIN
   WHILE [cur-step]is-not-a-return-step DO
      COMPUTE! field-routine-termination-in-destination-env
         ( TRY execute-steps-in-cur-routine, called-locus )
   RETURN-WITH-VALUE! NIL
END.

PROC-DF execute-steps-in-cur-routine
BEGIN
   WHILE [cur-step]is-not-a-terminator-step DO
      COMPUTE! effect-of-step( cur-step )
   RETURN-WITH-VALUE! terminator-destination
```

```
DF field-routine-termination-in-destination-env(dest-locus,called-locus)
=>  normal-return IF dest-locus EQS void-locus ;
=>  continue-at-destination-locus( dest-locus )
    IF env-of-locus(dest-locus) EQS env-of-locus(called-locus) ;
=>  finish-program-execution IF called-locus EQS program-called-locus ;
=>  FAIL-WITH-VALUE deallocate-cur-invocation(dest-locus) OTHERWISE .
```

A return-step is either a RETURN statement or a syntactic marker <end-body> placed at the end of each routine body, causing an implicit RETURN. A terminator-step is either a return-step, an ABORT statement (causing return of control to the explicit or inherited ABORT-destination for the current invocation), a STOP statement (causing abnormal termination of the program with a possible integer value returned to the system), or a GOTO statement whose destination-name is a formal-parameter label (causing abnormal termination of the current invocation and the effect of executing a GOTO to the actual-parameter label in the calling invocation). The effects of non-terminator steps are described later in this section.

During normal execution, execute-steps-in-cur-routine repeatedly determines the effect of the current executable step (identified in the global variable cur-step); this effect includes changing cur-step to indicate the next step to be executed, usually the next step in textual order in the sequence of steps for the current routine body. This repetition continues until cur-step indicates a terminator-step, whose effect will change cur-env by ending the current invocation. The locus of control to result from this change (the "destination locus") is constructed by terminator-destination; it becomes the value of execute-steps-in-cur-routine, and thus becomes the dest-locus argument of field-routine-termination-in-destination-env for the current invocation. This last DF may evaluate FAIL-WITH-VALUE to terminate the current invocation, passing the destination locus back as the preemptive value of execute-steps-in-cur-routine for the calling invocation; thus this "fielding DF" both generates and interprets all preemptive values used in specifying J73 control semantics.

- When dest-locus is the void-locus indicating a normal return from the current routine, the fielding DF has a NIL effect and returns to the remainder of invoke-routine, which assigns result-parameter values, saves any return value for a function, and deallocates the automatic data of the completed invocation.

- When dest-locus contains the environment of a prior routine invocation, the current invocation's instance of the fielding DF uses FAIL-WITH-VALUE to preempt the current invocation and supply dest-locus to the caller's instance of the fielding DF.

- When dest-locus contains the environment of the current invocation, the fielding DF resets the current locus to this destination and calls execute-steps-in-cur-routine to continue this invocation from the new cur-step.

- When dest-locus is the system-locus indicating abnormal program termination, the program-body's instance of the fielding DF must reset the current locus to the program-completion locus, using the normal-return mechanism to return to the "system", since there is no active "TRY execute-steps-in-cur-routine" to field a FAIL-WITH-VALUE at the system level; terminator-destination has already given a value to a distinguished program-value variable, to indicate abnormal termination and a possible STOP value.

As an example of this mechanism, consider the code

```
START PROGRAM PP; BEGIN
   QQ ABORT LL;  unexecuted text ; LL:  continuation text ...
   PROC QQ; BEGIN
      RR; unexecuted text ; ...
      PROC RR; BEGIN
         ABORT; unexecuted text ; ... END  END  END  TERM
```

In this J73 program, the program-body calls QQ with an abort-phrase; QQ's call to RR inherits LL as an abort-destination; and the ABORT statement in RR returns control to the label LL in the program-body. The history of pertinent DF-calls in the Specification's interpretation of this program is given below. (Environments and steps are represented symbolically for greater clarity in this discussion.)

1. execute-invoked-body for invocation #1 (system calls the program-body)
   calls execute-steps-in-cur-routine for program-body steps,
   which calls invoke-procedure for the call to QQ,
   passing the abort destination [#1, LL] t the invocation of QQ.

2. execute-invoked-body for invocation #2 (program-body calls QQ)
   calls execute-steps-in-cur-routine for steps in QQ,
   which calls invoke-procedure for the call to RR,
   passing the abort destination [#1, LL] to the invocation of RR.

3. execute-invoked-body for invocation #3 (QQ calls RR)
   calls execute-steps-in-cur-routine for steps in RR,
   which returns the inherited abort destination [#1, LL]
   as the value of terminator-destination, which becomes dest-locus for
   field-routine-termination-in-destination-env within execute-invoked-body #3.

4. This fielding DF for invocation #3 finds that its current environment #3
   is not the destination environment #1,
   so it uses FAIL-WITH-VALUE to preempt further execution in invocation #3,
   and to replace the evaluation of execute-steps-in-cur-routine #2
   with the destination locus, which becomes the dest-locus argument to
   the fielding DF for invocation #2.

5. The fielding DF for invocation #2 similarly finds that its current
   environment #2 is not the destination environment #1,
   so it uses FAIL-WITH-VALUE to preempt further execution in invocation #2,
   and to replace the evaluation of execute-steps-in-cur-routine #1
   with the destination locus, which becomes the dest-locus argument to
   the fielding DF for invocation #1.

6. The fielding DF for invocation #1 finds that its current environment is
   the destination environment #1, so it resets the current locus to
   environment #1 and step LL in the program-body, and
   execute-invoked-body #1 again calls execute-steps-in-cur-routine to
   continue execution from the label LL.

The semantics of statements that do not terminate the current routine invocation is specified by the DF effect-of-step. Part of the effect of each executable step is to update the global variable cur-step which identifies the next step to be executed. The steps below whose names have the form end-... are syntactic marker nodes that produce the NIL string of terminal characters in the Specification syntax for J73, and simplify the description of effect-of-step.

The effect of an assignment statement is

- first, to evaluate the constituent formula to get the representation of the value being assigned;
- then, for each variable, to determine the standard name designating that variable object and to use put-value to assign the value to that variable; and
- finally, to advance cur-step to the next step in the current routine body (the textual successor of the assignment statement).

The effect of a GOTO statement (whose destination-name is not a formal parameter) is to set cur-step to the defining-occurrence of its destination name. A label defining-occurrence is considered an executable step in order to simplify the description of effect-of-step; its effect is the trivial one of advancing cur-step to its textual successor step.

The effect of a procedure-call statement is described by invoke-procedure for user-defined procedures being invoked, or by specific implementation-dependent DFs for any machine-specific procedures. cur-step is set to its textual successor step upon normal return from the procedure-call; abnormal termination of the procedure-call preempts this effect.

The description of IF-statement effects utilizes the syntactic markers end-conditional-statement and end-if-statement, placed as shown below.

```
if-statement =:  if-clause  statement  end-conditional-statement
                          [ ELSE  statement ]  end-if-statement
```

The effect of an if-clause is to advance cur-step either to its own textual successor if its boolean-formula evaluates to TRUE, or to the successor of the corresponding end-conditional-statement step (thus executing any ELSE) otherwise. The effect of an end-conditional-statement step is to advance cur-step to the successor of the corresponding end-if-statement step (thus skipping any ELSE). The effect of an end-if-statement step is to advance cur-step to its textual successor.

The description of WHILE-statement effects utilizes the marker end-while-statement, placed as shown below.

```
while-statement =:  while-clause  statement  end-while-statement
```

The effect of a while-clause is to advance cur-step either to its own textual successor if its boolean-formula evaluates to TRUE, or to the successor of the corresponding end-while-statement otherwise.
The effect of an end-while-statement is to set cur-step to the corresponding while-clause.

The description of FOR-statement effects utilizes the marker end-for-statement, placed as shown below.

```
for-statement  =:  for-clause  statement  end-for-statement
for-clause     =:  FOR  control-item  :  control-clause  ;
control-clause =:  initial-value  [ while-phrase and/or
                                    by-or-then-phrase   ]
```

The effect of a for-clause is to assign the specified initial value to the control-item (using put-value), and then to advance cur-step to its textual successor.
The effect of a while-phrase is to advance cur-step to its textual successor if its boolean-formula evaluates to TRUE, or to deactivate the letter control-item (by giving it the value UNDEFINED) and advance cur-step to the successor of the corresponding end-for-statement otherwise.
The effect of an end-for-statement is to modify the current value of the loop control-item according to any by-phrase or then-phrase for the loop, and then to set cur-step to the textual successor of the corresponding for-clause.
Note that the sequence of executable steps in a body is constructed by a prefix tree walk of the parse tree for the body, so any constituent step within a step is a successor of that enclosing step. Thus the textual successor of a for-clause is the while-phrase, if any, in the for-clause, and is otherwise the statement to be iterated.

The effect of an EXIT statement is to advance cur-step to the textual successor of the end-for-statement or end-while-statement corresponding to the loop being exited, after deactivating any letter control-item for that loop.

The description of CASE-statement effects utilizes the markers end-case and end-case-statement, placed as shown below.

```
case-statement    =:  CASE  case-selector  ;  BEGIN  case-body
                      [ labels ]  END  end-case-statement
case-body         =:  [ ( DEFAULT ) :  statement  opt-fallthru ]
                      case-alternative ...
case-alternative  =:  ( case-indices ) : statement  opt-fallthru
opt-fallthru      =:  FALLTHRU or
                      end-case
```

The effect of a case-statement is to evaluate the case-selector formula and then to advance cur-step to the first step following the case selected by this value.

The effect of an end-case step is to advance cur-step to the textual successor of the corresponding end-case-statement.

The effect of an end-case-statement is to advance cur-step to its textual successor.

Implementation Dependencies

A distinct section of the Specification collects the DFs which attempt to encapsulate to some extent the various differences between implementations of J73. These DFs represent the following implementation-specific semantics.

- values of implementation parameters
  (current values are arbitrary but form a
  consistent set of parameter values)
- interpretation of !COPY directives to retrieve
  the text to be substituted
- choice of whether to implement lower-case letters
- interpretation of compool-file-name in !COMPOOL directives
  to identify a compool module
- default method of truncation of numeric values to
  a given precision
- character-set accepted, with collating sequence
- order of evaluation of actual parameters
- machine-specific procedures and functions provided by a system
  (currently limited to a set of names recognized as output
  procedures used by the Validator)
- forms of representation of values of a given type,
  especially numeric types, with definitions of J73 operations
  in terms of such representations
  (limitations of the current value representations are
  discussed in Representation of Values)

Summary of Specification Effort

Development of the SEMANOL Specification of JOVIAL(J73) was based on a careful analysis of MIL-STD-1589A as the definition of the J73 language; this analysis is described elsewhere in this report. Effort was then expended to design, develop and partially debug the Specification.

Design goals for the Specification included the usual goals common to most SEMANOL Specifications; the Specification should be readable, executable, complete, traceable to the MIL-STD definition, and protected against interpretation of illegal "programs" in the language being specified, to the extent feasible with the level of effort available. In addition, because the purpose of the Specification in this project was to partition the J73 language for the Measurement Facility's use, the project context determined certain other goals for the Specification design:

- separate syntactic DFs for grammatically-separable concepts
- separate semantic DFs for distinct concepts having similar syntax
- separate DFs for distinct variants of a construct in a given context using that construct, where feasible (for example, distinguishing the effect of a true from a false boolean-formula in an IF statement)
- use of DF names that are highly descriptive of DF meanings
- separation of compilation semantics from execution semantics, to allow for compile-only tests
- emphasis on producing a fairly-complete executable Specification (for a subset of "safe" J73 programs), rather than an unexecutable design for a complete Specification or an incomplete Specification that rejects all illegal "programs".

An example of how the separation goals affected the Specification design is the division of the effect of an IF statement into the two DFs effect-of-true-if-clause and effect-of-false-if-clause, even though these two DFs are so simple that they would be clearly readable as cases of a single DF effect-of-if-clause.

The emphasis on producing an executable Specification rather than a complete design was important, since the complexity of the J73 language precluded development of a complete executable Specification with the level of effort available. In order to support the Measurement Facility, the lexical analysis and context-free grammar accept all legal J73 programs, so that syntactic coverage can be assessed. Separation of distinct concepts is further achieved by allowing compile-time semantic DFs to distinguish concepts that run-time semantic DFs treat identically in the implemented models. For example, the implemented storage model is not sufficiently detailed to represent the exact semantics of the various kinds of specified tables; all are generally treated as if they were ordinary tables. Construction of the type of a specified table, however, differentiates these kinds of tables; thus, with the current Specification coverage can be assessed to the level of these different kinds of tables, although not to the level of assignment to each kind of table or parameter associations for each kind of table.

The Specification was then implemented according to its design, using the SEMANOL(76) metalanguage including the TRY and FAIL-WITH-VALUE operators. The J73 concepts whose detailed semantics are not precisely described by the current Specification are listed below.

1. The scope of a define-name is not correctly determined when the proper declaration of the name is not the textually most-recent declaration of that name.

Correction of omission 1 would require prohibitively-large computational-time resources using the encapsulated SEMANOL Parser, and is not recommended.

2. Explicit importation of a compool-declared-name does not cause the implicit importation of any other names.

3. Preset values in blocks and tables are not implemented.

4. LIKE options in table types are not implemented.

5. Defining-occurrences and declaring-occurrences of block instantiations may not be correctly linked.

6. Table assignment is implemented by reference semantics, and is thus incorrect when selection of a target table variable contains a function-call altering the stored value of the table-object in the assignment formula.

7. !TRACE directives are not implemented.

8. Various constraints on legal J73 programs are not enforced. (See Constraint Enforcement).

Concepts 2. through 8. were omitted in order to scope the Specification task according to the level of effort available; these particular concepts were chosen for omission in the semantic description because they are not part of the Validator's set of critical J73 features used to test other features. Their correct implementation would require additional time but would not make the Specification significantly more complex. The current Specification supports some coverage assessment for all but concept 2.

9. Some directives have no particular significance for the canonical form of the *SEMANOL* Specification, and are ignored in the semantic descriptions; they are distinguished syntactically (thus allowing assessment of coverage for these directives). These directives include

   - !LINKAGE, since the Specification does not represent specific linkage conventions, J73 or otherwise;
   - !INTERFERENCE, since the Specification does not perform any of the potentially-dangerous optimizations to be inhibited by this directive;
   - listing directives, since the Specification does not produce a paged source listing due to resource requirements;
   - register directives, since the Specification does not utilize registers;
   - expression evaluation order directives, since the Specification does not deviate from left-to-right evaluation of expressions;
   - INITIALIZE!, since the meaning of "all zero bits" as a value representation for nonbit-type values is not defined in the Specification.

10. No invertible relationship between integers and the pointer addresses for storage of data variables and compiled code is defined; conversions between pointers and integers are implemented by constant DFs, and pointer conversions and the NEXT function for pointers always yield the undereferenceable value NULL.

11. No invertible relationship between bitstring values and the underlying representations of all data values is defined; conversions between bit and nonbit types are implemented by constant DFs, and the REP pseudovariable is implemented using SEMANOL character-string underlying representations of values.

12. BITSIZE, and by derivation therefrom, BYTESIZE, WORDSIZE, and NWDSEN, of tight tables and packed tables containing filler bits, and of blocks containing such tables, do not reflect alignment restrictions for table components relative to word boundaries and byte boundaries; the Specification does not support all details of J73's view of storage as words consisting of fixed-length bitstrings.

13. Construction and use of aliases for data storage is not supported, including overlay-declarations, relative positioning of component objects in blocks and tables, REP pseudovariables storing values not of an expected type, NEXT for pointers (see 10. above), and dereference of a typed pointer converted from an integer (see 10. above) or from a differently-typed pointer.

Concepts 10. through 13. were omitted because they represent a very significant part of the complexity of a complete Specification of J73, and are not part of the Validator's set of critical features. Their correct implementation would require additional time and would impact a significant portion of the current Specification. Their correct implementation was well beyond the level of effort available. The current Specification supports some coverage assessment for these concepts.

Finally, the developed Specification was partially tested and debugged. The Specification now is translated by the SEMANOL Interpreter without error, and correctly executes a small collection of ad-hoc test programs (i.e. these programs were not taken from the Validator developed for this project). A few corrections to the Specification text given in Appendix B were identified during testing, and are given below. The results of the test programs for the Specification (with these corrections as incremental changes) follow the correction text.

"The following are replacements for DFs in Appendix B.
These corrections were included in the Specification used to
execute the tests reported in Appendix A."

```
#SEMANTIC-DEFINITIONS:


#DF is-an-expr (n)

     "{ n #IS #NODE }"

     => #TRUE #IF n #IS <formula> #U <relation> #U
        <relative>;

     => #TRUE #IF n #IS <term> #U <factor> #U <primary>;

     => #TRUE #IF parent(n) #IS <primary> #U <literal>
             #U <intrinsic-function> #U <pseudovariable>;

     => #TRUE #IF n #IS <loop-control-letter> #U
        <dereference>;

     => #TRUE #IF n #IS <ct-formula> #U <pointer-formula>;

     => #TRUE #IF n #IS <status-list-index> #U
        <item-preset-value>;

     => #TRUE #IF n #IS <lower-bound> #U <upper-bound>;

     => #TRUE #IF n #IS <entry-size> #U <overlay-address>;

     => #TRUE #IF n #IS <constant-index> #U
        <repetition-count>;

     => #TRUE #IF n #IS <boolean-formula> #U
        <case-selector-formula>;

     => #TRUE #IF n #IS <case-lower-bound> #U
        <case-upper-bound>;

     => #TRUE #IF n #IS <input-argument> #U
        <output-argument>;

     => #TRUE #IF n #IS <variable>;
```

```
     => #FALSE #OTHERWISE #.


#DF field-routine-termination-in-destination-env
           (dest-locus, called-locus)

     => normal-return #IF dest-locus #EQS void-locus;

     => continue-at-destination-locus(dest-locus)
           #IF env-of-locus(dest-locus) #EQS
        env-of-locus(called-locus);

     => finish-program-execution
           #IF env-of-locus(dest-locus) #EQS
        env-of-locus(system-locus);

     => #FAIL-WITH-VALUE
             deallocate-cur-invocation (dest-locus)
        #OTHERWISE #.


#DF seq-of-steps-in (b)

     => #SUBSEQUENCE-OF-ELEMENTS s #IN
           ( #SEQUENCE-OF  <end-body> #U
             <assignment-statement> #U <exit-statement>
             #U <stop-statement> #U <abort-statement>
             #U <goto-statement> #U
        <procedure-call-statement> #U
             <case-statement> #U <end-case> #U
        <end-case-statement> #U
             <for-clause> #U <while-phrase> #U
        <end-for-statement> #U
             <while-clause> #U <end-while-statement> #U
        <if-clause> #U
             <end-conditional-statement> #U
        <end-if-statement> #U
             <label-name> #IN b)
           #SUCH-THAT( body-of(s) #EQN body-of(b) ) #.



#DF body-of (s)

     => child( #SEG 7 #OF parent(s) ) #IF s #IS
```

AD-A102 386 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CA F/G 9/2
JOVIAL (J73) COMPILER VALIDATOR.(U)
JUN 81 R M HART, M S MCCLANAHAN F30602-79-C-0221
UNCLASSIFIED RADC-TR-81-128 NL

2 OF 2
AD A
102 386

END
DATE FILMED
9-81
DTIC

```
      <subroutine-name>;

   => s #IF s #IS <body>;

   => child(s) #IF s #IS <program-body> #U
      <subroutine-body>;

   => #LAST n #IN (#SEQUENCE-OF-ANCESTORS-OF s)
         #SUCH-THAT ( n #IS <body> ) #OTHERWISE  #.
```

The test results reported herein were obtained
as absentee Multics jobs under the following top ec :

```
& abs_test.absin     February 1981     McClanahan
&   execute a J73 test program as an absentee process
&   &1 = test program in j73test1981.archive
&command_line off
ready_off
string "= = = = = = = =" &1 "= = = = = = = ="
ac x j73test1981 &1
pr &1 1 200
string "- - - -" end of text "- - - -"
semanol sil.j73
load isil.j73
bron execution
string "- - - -" compilation "- - - -"
ready_on ;run &1
ready_off
string "- - - -" execution "- - - -"
ready_on;continue
ready_off
calst
dl &1
```

The following are test results showing correct interpretation
of J73 ad-hoc test programs (not tests from the Validator
developed for this projet) by the SEMANOL Specification of
JOVIAL(J73). They include sequence-values printed to check
the dynamic history of routine invocations and completions.

```
= = = = = = = = test.exec.1 = = = = = = = =
start program pp ; stop ; term
\ 1.2.3 minimal j73 program
  should compile, link and execute without error;
  terminate program with value not determined
- - - - end of text - - - -
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
```

STOP

```
STOP

STOP
- - - - compilation - - - -


Revised text of module 1:
START PROGRAM PP ; STOP ; TERM

End revised module text

successful compilation of module 1
break at execution
in #CONTROL at location 9 : level 1

STOP
r 1901 367.643 116.430 658

- - - - execution - - - -

program accepted

initial-program-locus:
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:35 \ :end-seq-dump

program-called-locus:
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:35 \ :end-seq-dump

program-completion-locus:
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:42 \ :end-seq-dump

dest-locus FOR field-term... :
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:1 \ :end-seq-dump

called-locus FOR field-term... :
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:35 \ :end-seq-dump

program-execution-terminated-with-value:NOT DETERMINED.
mstop called
in #CONTROL at location 11 : level 1

STOP
r 1901 5.047 1.116 4
```

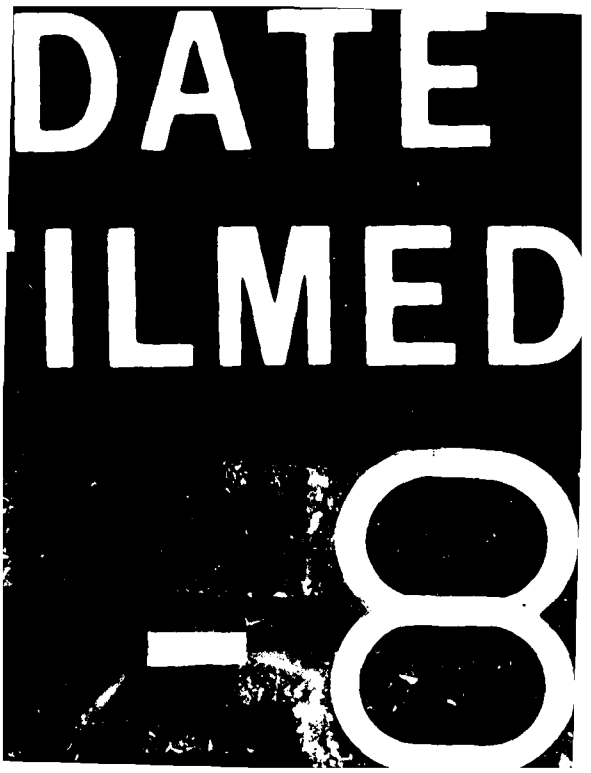
```
= = = = = = = = test.exec.2 = = = = = = = =
StarT program Pp; if tRuE ; WHILE false; STOP 1 ;TERm
\ 1.2.3 main-program-module
  4.2   while statement (no iteration)
  4.3   if statement (then-branch)
  8.1   equivalent cases of letters
  should compile, link and execute without error;
  complete program normally
- - - - end of text - - - -
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns

STOP

STOP
- - - - compilation - - - -


Revised text of module 1:
START PROGRAM PP; IF TRUE ; WHILE FALSE; STOP 1 ;TERM

End revised module text

successful compilation of module 1
break at execution
in #CONTROL at location 9 : level 1

STOP
r 1843 413.279 98.152 1658

- - - - execution - - - -

program accepted

initial-program-locus:
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:35 \ :end-seq-dump

program-called-locus:
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:35 \ :end-seq-dump

program-completion-locus:
```

```
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:112 \ :end-seq-dump

dest-locus FOR field-term... :
seq-dump:\
seq-dump:\  \ :end-seq-dump, node:1 \ :end-seq-dump

called-locus FOR field-term... :
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:35 \ :end-seq-dump

program-execution-completed.
mstop called
in #CONTROL at location 11 : level 1

STOP
r 1843 13.885 1.100 23



= = = = = = = = test.exec.3 = = = = = = = =
start program pp ; begin
  case 1B'1' ; begin
    (true): ; (false) : stop ; end end term
\ 4.4 case statement (no default)
  should compile, link, execute without error
  complete program normally
- - - - end of text - - - -
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns
scomp called
scomp returns

STOP

STOP

STOP
- - - - compilation - - - -


Revised text of module 1:
START PROGRAM PP ; BEGIN
CASE 1B'1' ; BEGIN
```

```
(TRUE): ; (FALSE) : STOP ; END END TERM

End revised module text

successful compilation of module 1
break at execution
in #CONTROL at location 9 : level 1

STOP
r 2108 445.341 141.702 1080

- - - - execution - - - -

program accepted

initial-program-locus:
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:47 \ :end-seq-dump

program-called-locus:
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:47 \ :end-seq-dump

program-completion-locus:
seq-dump:\
seq-dump:\ 0 \ :end-seq-dump, node:187 \ :end-seq-dump

dest-locus FOR field-term... :
seq-dump:\
seq-dump:\  \ :end-seq-dump, node:1 \ :end-seq-dump

called-locus FOR field-term... :
seq-dump:\
seq-dump:\ 0, 1 \ :end-seq-dump, node:47 \ :end-seq-dump

program-execution-completed.
mstop called
in #CONTROL at location 11 : level 1

STOP
r 2108 17.154 0.102 12
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DATE
ILMED
8